UNIVERSIDADE FEDERAL DO PARANÁ

APLICAÇÃO DE *DESIGN PATTERNS* E *LIGHTWEIGHT FRAMEWORKS* NO DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETOS REUTILIZÁVEIS: S.C.O. – SISTEMA DE CONTROLE DE OCORRÊNCIAS

CURITIBA
2007

FABRÍCIO SILVA KYT

RODRIGO VINÍCIUS PERLY

APLICAÇÃO DE *DESIGN PATTERNS* E *LIGHTWEIGHT FRAMEWORKS* NO DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETOS REUTILIZÁVEIS: S.C.O. – SISTEMA DE CONTROLE DE OCORRÊNCIAS

Trabalho de conclusão de curso apresentado à banca examinadora do Curso Superior de Tecnologia em Informática da Universidade Federal do Paraná como exigência parcial para obtenção do grau de Tecnólogo em Informática.

Orientador: Prof. Dr. Mauro José Belli.

CURITIBA
2007

FABRÍCIO SILVA KYT
RODRIGO VINÍCIUS PERLY

APLICAÇÃO DE DESIGN PATTERNS E LIGHTWEIGHT FRAMEWORKS NO DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETOS REUTILIZÁVEIS: S.C.O. – SISTEMA DE CONTROLE DE OCORRÊNCIAS

Este trabalho de conclusão de curso foi julgado adequado à obtenção do grau de Tecnólogo em Informática e aprovado em sua forma final pelo Curso Superior de Tecnologia em Informática da Universidade Federal do Paraná.

Curitiba, 07 de maio de 2007.

Banca Examinadora:

_____

Professor Dr. Mauro José Belli
Orientador

_____

Professor Dieval Guizelini
Integrante da Banca Examinadora

_____

Professora Dra. Jeroniza Nunes Marchaukowski
Integrante da Banca Examinadora

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. [GAMMA, 1995]

RESUMO

No processo de desenho de aplicações devemos ao mesmo tempo ser específicos para solucionar o problema proposto e o suficientemente genéricos para integrar futuras implementações. Esta condição é que torna extremamente difícil o desenvolvimento de softwares orientados a objetos que sejam reutilizáveis. A arquitetura da aplicação deve ser precisamente planejada e a interação entre os componentes controlada, de forma a reduzir dependências entre eles.

Este trabalho apresenta a adoção de uma série de padrões de desenho e *ligthweigth frameworks* durante o desenvolvimento de uma aplicação orientada a objetos reutilizável para o controle de ocorrências internas e externas de uma companhia.

Palavras-chave: *lightweight frameworks*, orientação a objetos, *softwares* reutilizáveis, *design patterns*, *MVC*, Modelo 2 *Sun, Java*, *Struts*, *Hibernate*.

ABSTRACT

When developing applications our design should be specific to the problem at hand but also general enough to address future requirements. This requisite makes the development of reusable object-oriented software extremely hard. The application architecture must be precisely planned and the components interaction controlled to prevent dependencies.

This essay presents the adoption of various design patterns and lightweight frameworks during the development of a reusable object-oriented application that manages internal and external occurrences of a company.

Keywords: lightweight frameworks, object-oriented software, reusable softwares, design patterns, MVC, Sun´s Model 2, Java, Struts, Hibernate.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

1 INTRODUÇÃO

Com o crescimento da indústria de tecnologia da informação o mercado necessita, cada vez mais, da criação de softwares complexos, com custos estimáveis, em períodos de tempo curtos, utilizando-se dos mais variados avanços tecnológicos disponíveis e de padrões de desenvolvimento de alta qualidade. Isto torna a implementação necessária para a criação, manutenibilidade e reutilização de um software uma tarefa extremamente complexa, aumentando o esforço de desenvolvimento.

Um fator responsável por este aumento é a escassez na documentação dos problemas enfrentados por desenvolvedores e suas respectivas soluções. Sendo assim, situações que muitas vezes se repetem, acabam por gerar esforços adicionais de implementação. Da mesma forma, as tentativas de reuso destas soluções e o acúmulo de experiências sobre determinados problemas são iniciativas isoladas de pequenos grupos de desenvolvedores.

Outro fator responsável pelo aumento do esforço no desenvolvimento de uma aplicação é a não utilização de *frameworks*. Muitos desenvolvedores encontram-se temerosos a utilização de *softwares* reutilizáveis que não tenham sido produzidos dentro da própria corporação. Isto aliado ao impacto no aprendizado necessário para a utilização de um framework contribui para seu desuso.

Baseado nestes dois fatores, este trabalho abordará a aplicação e as vantagens na utilização de *lightweight frameworks* e *design patterns* no desenvolvimento de uma aplicação orientada a objetos para o controle de ocorrências internas e externas de uma companhia, levando em consideração os princípios da facilidade de manutenção, portabilidade, rastreabilidade, reusabilidade e modularidade.

1.1 Objetivos Gerais

O objetivo deste projeto é aplicação de *design patterns* e *ligthweight frameworks* no desenvolvimento de *softwares* orientados a objetos reutilizáveis, tendo como produto final uma solução *web* para o registro de ocorrências internas e externas de uma companhia.

1.2 Objetivos Específicos

Os métodos e matérias utilizados no desenvolvimento do projeto, tantos quantos forem possíveis, deverão ser *softwares* livres e de código-fonte aberto.

A aplicação deverá seguir o paradigma de programação orientado a objetos, utilizando desta forma suas características e melhores recomendações.

Aliados a estas condições deverão ser levados em consideração os princípios da facilidade de manutenção, portabilidade, rastreabilidade, reusabilidade e modularidade.

1.3 Justificativa

A CELEPAR – Companhia de Informática do Paraná é responsável por prover os serviços de informática para os órgãos administrativos do Governo do Estado do Paraná.

Foi desenvolvida por esta companhia uma aplicação aparentemente genérica para o controle das demandas de serviço e atendimento das ocorrências internas e externas de seus clientes, como denúncias, sugestões e dúvidas. Esta aplicação foi desenvolvida em Lotus Notes, uma tecnologia proprietária da IBM, que possui vários problemas já diagnosticados pela comunidade de desenvolvedores. Além disso, a aplicação possui uma quantidade significativa de falhas graves em seu desenvolvimento e operação.

Em seu desenvolvimento não foram tomados os devidos cuidados para que a arquitetura da aplicação tornasse viável o suporte à utilização por mais de um

órgão. Para tratar esta condição a solução adotada pela CELEPAR foi gerar uma cópia da aplicação (base notes) para cada órgão que desejasse fazer uso do serviço, o que criou problemas de manutenção e impossibilitou a integração de dados entre as bases.

Durante sua operação não são raras as situações em que o sistema duplica uma ocorrência que é enviada a um grupo de técnicos e mais de um profissional a soluciona, gerando desperdício de tempo e recursos.

Ocorrem também, exclusões de ocorrências que não foram solicitadas, sendo necessário que o *datacenter* da CELEPAR restaure *backups* constantemente.

Durante os atendimentos telefônicos, constantes travamentos da aplicação impossibilitam o preenchimento dos dados da ocorrência de forma rápida, fazendo com que esta atividade crítica do processo tenha seu desempenho degradado.

Várias ocorrências não são encaminhadas de forma correta, ficando sem atendimento até que, de forma manual, algum atendente verifique o erro ou os administradores recebam avisos de prazo de atendimento ultrapassado.

As consultas às ocorrências são insatisfatórias, pois não permitem ao atendente localizar as ocorrências abertas no mesmo dia, trazendo apenas as ocorrências abertas do dia anterior para trás.

As constantes interrupções na acessibilidade aos servidores Domino[1] e a baixa prioridade que a CELEPAR passou a conceder a manutenção dos sistemas desta plataforma é fator agravante, já que a partir do momento no qual o Lotus Notes deixou de ser a ferramenta oficial de correio eletrônico da companhia existe a intenção de desativar seu uso para as demais atividades.

Em decorrência destes fatores foi identificada a necessidade do desenvolvimento de um novo aplicativo que substitua o anterior e agregue novas funcionalidades.

Porém, foram estabelecidos, pelo cliente, como requisitos da aplicação a utilização dos *lightweight frameworks Struts* e *Hibernate*. Esta opção demandou a seleção de diversos *design patterns* que neste projeto possuem o intuito de

[1] Servidor de aplicação no qual ficam hospedadas as aplicações desenvolvidas em *Lotus Notes*.

auxiliar o desacoplamento da aplicação com os *frameworks*, disponibilizando uma solução única, expansível e adaptável.

Para atender as novas diretrizes do Governo do Estado do Paraná o sistema deve ser desvinculado de tecnologias proprietárias.

2 Metodologia

Os métodos e materiais englobam todas as técnicas e ferramentas utilizadas durante a realização do projeto. Elas podem ser divididas em três categorias: planejamento, modelagem e implementação.

2.1 Metodologia de Planejamento

O planejamento do projeto foi composto pelo plano de projeto e pelos relatórios mensais de acompanhamento.

Os relatórios mensais foram utilizados como protocolo de comunicação entre a equipe de desenvolvimento e o gerente de projeto. O intuito dos relatórios foi gerar um mecanismo padronizado para o acompanhamento documentado da realização das tarefas do projeto.

O plano de projeto procurou constituir o detalhamento dos objetivos do projeto, análise de riscos, cronograma, estimativas, organização da equipe bem como as principais funções desempenhadas pela aplicação.

2.1.1 Definição dos Objetivos

Para subsidiar a definição dos objetivos do projeto, foram realizadas reuniões com o analista de negócio[2], nas quais foram traçados os requisitos funcionais e elucidadas as expectativas do desenvolvimento da nova solução.

[2] Analista de negócio é um profissional, da área de informática ou não, especialista no escopo de um problema.

2.1.2 Análise e Gerenciamento de Riscos

A metodologia para análise de riscos iniciou-se com o levantamento dos possíveis riscos associados ao andamento do projeto. Estes riscos foram projetados em graus de possibilidade de ocorrência e classificados quanto ao impacto no processo. Feito isso a etapa de gerência dos riscos buscou a elaboração de um plano de contingência caso algum risco tornasse a ser um problema real.

2.1.3 Recursos

A metodologia para o levantamento dos recursos pressupôs a divisão destes em 3 (três) grupos distintos: recursos de pessoal, recursos de *hardware* e *software* e recursos especiais.

Os recursos de pessoal foram moldados utilizando como base os papéis necessários para a realização das atividades de um projeto, como analistas de suporte, analistas juniores, analistas plenos, analistas de treinamento, analistas de negócio, gerente de projetos, administrador de dados, analistas de banco de dados, documentadores, homologadores, implantadores e programadores [QUADROS,2002].

Os recursos de *hardware* e *software* foram determinados pelo cliente e validados pela equipe de desenvolvimento que os dividiu em 4 (quatro) ambientes: banco de dados de produção, banco de dados de desenvolvimento, servidor de aplicação de desenvolvimento e servidor de aplicação de produção.

Os recursos especiais abrangeram todos aqueles que não puderam ser classificados em nenhuma das duas propostas acima.

2.1.4 Organização de Pessoal

A organização de pessoal foi realizada acerca do proposto nos recursos de pessoal, utilizando como critério para a designação pessoa-papel a aptidão na área de informática de cada membro da equipe de desenvolvimento.

2.1.5 Estimativas

Para a realização das estimativas foram utilizadas as técnicas de pontos por função *(function points)* [ALB,1979] e linhas de código *(lines of code)*. As duas técnicas apresentam abordagens distintas e foram utilizadas em conjunto para prover dois pontos de vista sobre o esforço necessário para o desenvolvimento da aplicação Inicialmente os requisitos funcionais da aplicação foram estimados através da técnica de pontos por função, permitindo que o esforço para o desenvolvimento da aplicação fosse medido de forma independente as tecnologias empregadas. Na seqüência os pontos por função foram convertidos em linhas de código Java, com o auxílio de dados históricos disponibilizados pela revista técnica *Software Development* de Outubro de 2000.

2.1.5.1 Métrica de Pontos por Função

De acordo com a técnica de análise de pontos por função, uma aplicação de software, vista sob a ótica do usuário, é um conjunto de funções ou atividades do negócio que o beneficiam na realização de suas tarefas. O manual do *IFPUG*[3] classifica os seguintes tipos de elementos funcionais:

- Entrada Externa – *EI (External Input)* – transações lógicas nas quais os dados entram na aplicação e mantém dados internos.
- Saída Externa – *EO (External Output)* – transações lógicas nas quais os dados saem da aplicação para fornecer informações para usuários da aplicação.
- Consulta Externa – *EQ (External Query)* – transações lógicas nas quais uma entrada solicita uma resposta da aplicação.
- Arquivos Lógicos Internos – *ILF (Internal Logical File)* – grupo lógico de dados mantido pela aplicação.

[3] Os conceitos sobre Pontos de Função foram inicialmente introduzidos por Allan Albrecht da *IBM*, em uma conferência da *Guide/Share* em 1979. Posteriormente, esses conceitos foram refinados em uma metodologia formal e publicados no domínio público em 1984. Subseqüentemente, uma comunidade de ávidos usuários resolveu efetuar padronizações adicionais nas regras de contagem de pontos de função, sendo formado o Grupo Internacional de Usuários de Pontos de Função (*IFPUG*), como um grupo formalmente constituído e sem finalidades lucrativas, em 1986. Desde então o *IFPUG* tem sido líder no estabelecimento e publicação de documentos relacionados a Pontos de Função, incluindo o Manual de Práticas de Contagem (*CPM*), atualmente na versão 4.0

- Arquivos de *Interface* Externa – *EIF (External Logical File)* – grupo lógico de dados referenciado pela aplicação, mas mantido por outra aplicação.

O manual do *IFPUG* fornece tabelas e diretrizes para determinar a complexidade de cada elemento funcional. Os elementos funcionais identificados são totalizados para calcular obtenção dos pontos por função não ajustados.

Então é calculado a partir de 14 (quatorze) características gerais dos projetos, que permitem uma avaliação geral da funcionalidade da aplicação: comunicação de dados, processamento distribuído, atualização de dados *online*, entrada de dados *online*, volume de transações, eficiência do usuário final, complexidade do processamento, facilidade de implantação, multiplicidade de locais, facilidade de mudanças, facilidade operacional, desempenho, utilização do equipamento e reutilização de código. A cada característica será atribuído um peso de 0(zero) a 5(cinco), de acordo com o nível de Influência na aplicação. O nível de Influência geral é obtido pelo somatório do nível de influência de cada característica e o fator de ajuste é obtido pela expressão:

Fator de ajuste = 0,65 + (nível de influência geral * 0,01)

O total de pontos por função da aplicação será encontrado mediante a multiplicação do número de pontos por função não ajustados pelo fator de ajuste:

PFs ajustados = PFs não ajustados * fator de ajuste

2.1.5.2 Métrica de Linhas de Código

A métrica de linhas de código consiste na contagem das linhas de código de uma aplicação. Embora esta métrica possa parecer simples, existe discordância sobre o que constitui uma linha de código. Para a maioria dos pesquisadores, a métrica de linhas de código não deveria contar linhas de comentário e linhas em branco, uma vez que estas servem para a documentação interna do programa e não afeta a sua funcionalidade. Um outro problema é que este sistema de medidas está fortemente ligado à linguagem de programação

utilizada, impossibilitando a utilização de dados históricos para projetos que não utilizam à mesma linguagem.

Este tipo de métrica é mais utilizado para a obtenção de informações de realização do projeto – quando este já está concluído –, sendo muito difícil o seu uso em estimativas.

2.1.6 Cronograma

O cronograma é um instrumento de planejamento e controle semelhante a um diagrama, no qual são definidas e detalhadas minuciosamente as atividades a serem executadas durante um período estimado de tempo.

Gerencialmente, um cronograma é um artefato de controle importante para levantamento dos custos de um projeto e, a partir deste artefato, pode ser feita uma análise de viabilidade antes da aprovação final para a realização do projeto.

Neste projeto foram utilizadas três técnicas que auxiliam na concepção, montagem e acompanhamento de cronogramas, o *WBS*, gráfico de *Gantt* e a rede de tarefas, respectivamente.

2.1.6.1 WBS (Work Breakdown Structure)

O diagrama *Work Breakdown Structure (WBS) é* uma ferramenta de decomposição do trabalho de um projeto em partes manejáveis. É estrutura em árvore exaustiva, hierárquica (do mais geral para o mais específico) de tarefas que precisam ser realizadas para completar um projeto. O objetivo de um *WBS* é identificar elementos terminais (itens reais a serem executados em um projeto). O *WBS* foi desenvolvido para ser completo, organizado e pequeno o suficiente para que o progresso possa ser medido, mas não detalhado o suficiente para se tornar, ele mesmo, um obstáculo para a realização do projeto. Desta forma ela pode ser usada como entrada para o desenvolvimento da agenda, atribuir funções ou responsabilidades, gerenciar riscos, entre outros. A *WBS* serve como base para a maior parte do planejamento de projeto.

2.1.6.2 Gráfico de Gantt

O gráfico de *Gantt* é um diagrama através do qual é possível organizar as tarefas de um projeto em relação ao tempo. Cada tarefa e representada na forma de uma barra. O tamanho das barras representa à duração das tarefas em dias. É possível estabelecer interdependências de tarefas informando antecessoras e sucessoras. Cada tarefa permite a alocação dos recursos humanos para a sua realização e o acompanhamento da porcentagem de finalização. A relação de dependência entre as tarefas permite a projeção do caminho crítico cujo acompanhamento é crucial para evitar atrasos na entrega do projeto.

2.1.6.3 Rede de Tarefas

A rede de tarefas é um diagrama para representação da inter-dependência de tarefas. Ele exibe como as tarefas cooperam para a conclusão do projeto. A rede é produto direto do gráfico de *Gantt.*

2.1.6.4 Software para Gerenciamento de Projetos: MS Project

O *Microsoft Project* é uma ferramenta profissional pra gerência de projetos. Ela possibilita o acompanhamento do projeto na maioria dos processos, como iniciação, planejamento, execução, controle e encerramento. Esta ferramenta permite aplicar técnicas avançadas de gestão de projetos, como gráficos de *Gantt*, rede de tarefas entre outras. Ela é ideal para aplicação das técnicas previstas no *PMBok*[4].

2.2 Metodologia de Modelagem

A metodologia de modelagem foi dividida em dois aspectos: a modelagem orientada a objetos (responsável por promover uma visualização clara e eficiente

[4] *Project Management Body of Knowledge (PMBOK)* é um padrão de gerência de projetos desenvolvido pelo *Project Management Institute (PMI)*.

do mundo real de modo a facilitar o desenvolvimento, implementação e manutenção da aplicação) e a modelagem de dados (responsável por promover uma visão da estrutura dos dados da aplicação).

2.2.1 Modelagem Orientada a Objetos

A modelagem orientada a objetos foi realizada utilizando a *UML,* uma linguagem para especificação, documentação, visualização e desenvolvimento de sistemas orientados a objetos. Seus diagramas representam a aplicação em diferentes níveis de abstração e sob diversas perspectivas. Por ser padronizada e possuir um vocabulário de fácil entendimento ela facilita a comunicação de todas as pessoas envolvidas no processo de desenvolvimento de *software*

A *UML* define duas categorias de diagramas: os diagramas estruturais e os diagramas comportamentais. Os diagramas estruturais servem para visualizar, especificar, construir e documentar os aspectos estáticos de uma aplicação, ou seja, a existência e a colocação de itens como classes, *interface*s, colaborações, e componentes. Os diagramas comportamentais, de forma contrária, são utilizados para visualizar, especificar, construir e documentar os aspectos dinâmicos de uma aplicação.

Os diagramas utilizados para a modelagem de partes estáticas de uma aplicação são: diagrama de classes, objetos, componentes e implantação. Enquanto os diagramas de caso de uso, interação (seqüência e colaboração), estados e atividades são dedicados à modelagem de partes dinâmicas.

Neste projeto, para a realização da modelagem orientada a objetos, foram utilizados os diagramas casos de uso, diagrama de classes e diagramas de seqüência.

2.2.1.1 Diagrama de Casos de Uso

O diagrama de casos de uso é utilizado para identificar os serviços externamente visíveis que a aplicação disponibiliza. Seu foco é estabelecido no que a aplicação deverá fazer, e não em como a aplicação irá fazer. Desta forma o

diagrama de casos de uso foi utilizado para realizar a modelagem do contexto e dos requisitos funcionais da aplicação.

O diagrama de casos de uso do SCO foi concebido através de incessantes reuniões com o analista de negócio, que forneceu as informações necessárias para a delimitação dos atores participantes do escopo da aplicação e o relacionamento destes com as funcionalidades que a aplicação deveria desempenhar.

2.2.1.2 Diagrama de Classes

O diagrama de classes é composto das classes, *interface*s, colaborações, relacionamentos de dependências, generalizações e associações presentes na aplicação. Suas principais utilidades são:

- Geração de um vocabulário capaz de definir o modelo de domínio da aplicação, o que torna possível a decisão de quais abstrações fazem parte do sistema e quais estão fora de seus limites;
- Identificação da relação entre um conjunto de classes, *interface*s e outros elementos que funcionam em conjunto proporcionando algum comportamento cooperativo;
- Ser utilizado como base para a modelagem dos dados, possibilitando a visão de como eles serão armazenados.

Para o desenvolvimento deste diagrama, uma profunda análise da descrição dos casos de uso permitiu que fossem levantadas as classes envolvidas no escopo da aplicação e seus métodos. Após este levantamento as classes foram estruturadas através da análise de suas colaborações, relacionamentos de dependência, generalizações e associações. Por fim, os atributos das classes foram adquiridos através de imagens das telas do sistema Lótus Notes em produção fornecidas pelo analista de negócios.

2.2.1.3 Diagramas de Seqüência

Os diagramas de seqüência permitem representar a interação formada por um conjunto de classes e seus relacionamentos. Através deles é possível visualizar claramente a existência dos objetos em um determinado período de tempo, no qual estes estão desempenhando uma ação ou trocando uma mensagem – diretamente ou por meio de um procedimento subordinado.

De uma forma geral os diagramas de seqüência foram utilizados para modelar aspectos dinâmicos da aplicação fornecendo subsídios para os processos reais a serem implementados. Programadores dotados de tais diagramas são aptos a programar as funcionalidades sem que seja necessário um conhecimento completo aplicação.

Estes diagramas foram produzidos com base na tradução realizada pelos analistas de sistemas dos casos de uso em operações informatizadas.

2.2.1.4 Software para Modelagem Orientada a Objetos: Jude

O *JUDE (Java and UML Developers Environment)* é uma *IDE* para modelagem funcional de sistemas*,* escrita em *Java,* de código-fonte aberto e de uso fácil e intuitivo. Com o *JUDE* é possível realizar uma modelagem complexa através da integração que a ferramenta possibilita entre os diagramas. A *IDE* suporta o desenvolvimento de 8 tipos de diagramas *UML*, como diagramas de classes, caso de uso, seqüência, atividade, colaboração e etc. O resultado da modelagem pode ser exportado para arquivos *Java*, *HTML* ou imagem.

2.2.2 Modelagem de Dados

A modelagem de dados é o desenho lógico de como os dados serão armazenados dentro de um banco de dados. Esta metodologia consiste na criação de tabelas e relacionamentos, bem como a definição de regras de integridade.

A principal ferramenta da modelagem de dados é o diagrama entidade relacionamento.

2.2.2.1 Diagrama Entidade Relacionamento

A modelagem dos dados foi realizada usando o diagrama entidade relacionamento (DER). Este diagrama descreve, em um alto nível de abstração, o armazenamento dos dados de uma aplicação em sistemas gerenciadores de bancos de dados. Sua aplicação reside em visualizar o relacionamento entre tabelas de um banco de dados, no qual as relações são construídas através da associação de um ou mais atributos destas tabelas

Este diagrama é baseado nas entidades persistentes do diagrama de classes. Uma série de análises e constatações permitiu a escolha das estratégias necessárias para a resolução do paradigma objeto-relacional na montagem das tabelas do banco de dados.

2.2.2.2 Software para Modelagem de Dados: DBDesigner

O *DBDesigner* é um *software* livre, multiplataforma, distribuído sob a licença *GPL*. É um editor visual para a criação de banco de dados que integra as tarefas de criação, modelagem de dados, desenvolvimento e manutenção em um ambiente simples e agradável. Suas principais características incluem:

- Suporte a bancos de dados que suportem acesso via *ODBC*;
- Engenharia reversa, gerando o modelo a partir das tabelas do banco de dados;
- Sincronia do banco de dados com as alterações realizadas no diagrama entidade relacionamento;
- *Interface* descomplicada;
- Salvar os arquivos em *XML*;
- Gerar relatórios em *HTML*.

2.3 Metodologia de Implementação

A metodologia de implementação foi inteiramente baseada na arquitetura cliente-servidor. Um servidor é um sistema de computação que fornece serviços a

uma rede de computadores. Esses serviços podem ser de diversas naturezas, como por exemplo, *ftp*, conexão remota e correio eletrônico. Os computadores que acessam os serviços de um servidor são chamados clientes.

Seguindo estas definições a metodologia de implementação foi dividida em: recursos lado cliente e recursos lado servidor.

2.3.1 Recursos lado Cliente *(Client Side)*

Os recursos lado cliente são todos aqueles que deverão ser processados ou executados por um computador cliente, ou seja, pelo computador que está acessando um determinado serviço de uma máquina servidora.

2.3.1.1 Páginas Web Estáticas: HTML e W3C Web Standards

A W3C. *(World Wide Web Consortium)* é um consórcio de empresas de tecnologia, fundado por Tim Berners-Lee em 1994, com o intuito de desenvolver w*eb standards* para a criação e interpretação dos conteúdos para a w*eb*.

Web Standards é um conjunto de normas, diretrizes, recomendações, notas, artigos, tutoriais e afins de caráter técnico, produzidos pelo *W3C* e destinados a orientar fabricantes, desenvolvedores e projetistas para o uso de práticas que possibilitem a criação de uma *web* acessível a todos, independentemente dos dispositivos usados ou de suas necessidades especiais.

Sites desenvolvidos segundo esses padrões podem ser acessados e visualizados por qualquer pessoa ou tecnologia, independente do *hardware* ou *software* utilizado.

Apesar da *W3C* não ser muito difundida na comunidade de desenvolvedores brasileiros, padrões seus como *HTML, XHTML* e *CSS* são extremamente populares.

O *HTML (HyperText Markup Language)* é uma linguagem de marcação utilizada pra a criação de páginas *web*. Ela permite que várias páginas se relacionem através de referencias chamadas *hyperlinks*. Através dela é possível

combinar textos, imagens e áudios, de forma que estes possam ser apresentados em navegadores de internet.

Entretanto, o *HTML* é uma linguagem extremamente limitada e suas deficiências tiveram de ser superadas através da introdução de outras tecnologias como *CSS* e *Javas*cript.

2.3.1.2 Formatação de páginas Web: Cascading Style Sheets

O *CSS (Cascading Style Sheets)* é uma tecnologia utilizada para a formatação de páginas *web*, cuja principal característica é prover a separação entre a formatação e o conteúdo de uma página.

Uma regra *CSS* é uma declaração que segue uma sintaxe própria e que define como será aplicado estilo a um ou mais elementos *HTML* .Um conjunto de regras *CSS* forma uma folha de estilos *(stylesheet)*. Uma regra *CSS*, na sua forma mais elementar, compõe-se de três partes: um seletor, uma propriedade e um valor.

O seletor é o elemento *HTML* identificado por sua *tag*, classe ou *ID* para qual a regra será válida (por exemplo: <p>, .minhaclasse e #divCabecalho).

A propriedade é o atributo do elemento *HTML* ao qual será aplicada a regra (por exemplo: *font, color, background* e etc).

O valor é a característica específica a ser assumida pela propriedade (por exemplo: *font: arial, color: blue, background: green*).

Os estilos podem ser vinculados a um documento de três maneiras distintas: *inline*, *embedded* e *external*.

Um *CSS* é dito externo *(external)*, quando as regras de estilo estão declaradas em um documento a parte do documento *HTML* – em uma folha de estilos. Uma folha de estilo externa é ideal para ser aplicada a várias páginas, pois permite que a aparência de um site esteja baseada em um único local. O arquivo da folha de estilos deve possuir a extensão *.css* e deverá ser referenciado no documento *HTML*. A sintaxe para criação desta referência é mostrada no quadro abaixo:

```
<html>
<head>
<link rel="stylesheet" type="text/css" ref="caminhodafolhadeestilos.css">
</head>
<body>
</body>
</html>
```

Quadro 1: Exemplo da sintaxe para inserção de uma referência a uma folha de estilos externa.

Um *CSS* é definido como incorporado *(embedded)* quando as regras de estilo estão declaradas no próprio documento *HTML*. Esta modalidade é ideal para ser aplicada a uma única página, pois com ela é possível mudar a aparência de somente um documento. A sintaxe é descrita abaixo:

```
<html>
<head>
<style type="text/css">
body {
        background: #000000;
        url("imagens/minhaimagem.gif");
}

h3 {
        color: #FF0000;
}

p {
        margin-left: 15px;
        padding:1.5em;
}
</style>
</head>
<body>
</body>
</html>
```

Quadro 2: Exemplo da sintaxe para inserção de um estilo incorporado.

Um *CSS inline* é caracterizado pela declaração das regras de estilo dentro da *tag* do elemento *HTML*. Esta metodologia perde as vantagens que o *CSS* disponibiliza, pois mistura o conteúdo com a apresentação. Este método deve ser usado excepcionalmente quando for necessário aplicar um estilo a uma única ocorrência de um elemento. A sintaxe para aplicar estilos *inline* é mostrada a seguir:

```
<html>
<head>
</head>
<body>
<p style="color:#000000; margin: 5px;">
</p>
</body>
</html>
```

Quadro 3: Exemplo da sintaxe para inserção de um estilo *inline*.

2.3.1.3 Processamento no cliente: Javascript

O *Javascript* é uma linguagem *script* desenvolvida para possibilitar o processamento de dados no lado cliente (navegador de *internet*). É uma linguagem fracamente tipada, baseada em programação prototipada e com sintaxe similar a linguagem C. Ela é comumente utilizada para tratar eventos sem a necessidade de realizar uma requisição ao servidor. Para utilizar o código *javascript* em uma página *html*, basta inserir a lógica de programação entre os marcadores *(tags)* <*script*> e <*/script*>. O *javascript* pode interagir com o *DOM*[5] *(Document Object Model)* da página em que esta sendo executado, o que permite a manipulação de qualquer elemento dentro do documento *HTML*.

2.3.1.4 Tableless

Tableless é uma metodologia para desenvolvimento de páginas *web* sem a utilização indiscriminada de tabelas. Esta metodologia defende que as marcações *HTML* devem ser utilizadas para o propósito o qual foram concebidas, sendo as tabelas criadas apenas para exibir dados tabulares e não para a montagem da estrutura de páginas.

Toda a formatação da página é feita através de arquivos *CSS*, o que torna a página mais leve e, portanto mais rápida de ser acessada.

A utilização do *tableless* é fortemente ligada aos *web standards* definidos pela *W3C*, o que torna necessário ao desenvolvedor possuir um excelente

[5] *Document Object Model (DOM)* e um padrão de representação de modelos de objetos baseados nos formatos *HTML* ou *XML* independente de plataforma ou linguagem.

conhecimento em *HTML/XHTML/CSS* para desfrutar de todas as vantagens da metodologia .

Algumas vantagens da utilização do *tableless* são:

- O desenvolvimento de páginas *web* melhor interpretadas pela maioria dos navegadores do mercado.
- Montagem de uma estrutura de HTML até 70% mais leve que a abordagem tradicional.
- Multiplataforma, pois as páginas podem ser acessadas em dispositivos móveis sem necessidade de versão especial.
- Melhoria na acessibilidade, porque possibilita que sejam programados recursos para facilitar o acesso por deficientes visuais.

2.3.1.5 Editor de Páginas Web: Dreamweaver

O *Macromedia Dreamweaver* é uma ferramenta proprietária da *Adobe Systems* para o desenvolvimento de páginas *web*. Versões iniciais da aplicação serviam como um simples editor *HTML WYSIWYG ("What You See Is What You Get")*[6], porém recentemente ela tem incorporado suporte a muitas outras tecnologias *web*, tais como *XHTML, CSS, Javascri*pt e alguns scripts servidor.

Como um editor *WYSIWYG*, o *Dreamweaver* esconde os detalhes do código *HTML* do usuário final, tornando possível que não-especialistas criem facilmente páginas *web*. Da mesma forma, ele oferece a usuários avançados amplo suporte aos *W3c's web standards*

O *Dreamweaver* também oferece integração com diversos navegadores de internet do mercado e facilita a gestão de *sites* internet.

Um aspecto interessante da arquitetura do *Dreamweaver* é a capacidade de inserção de extensões - pequenos programas anexados ao *software* principal que proporcionam funcionalidades adicionais.

---

[6] *WYSIWYG* é o acrônimo da expressão em inglês *"What You See Is What You Get"*, cuja tradução remete a algo como "O que você vê é o que você tem". Significa a capacidade de um programa de computador de permitir que um documento, enquanto manipulado na tela, tenha a mesma aparência de sua utilização. O uso inicial do termo foi relacionado a editores de texto, agora porém é aplicado a qualquer tipo de programa.

2.3.2 Recursos lado Servidor *(Server Side)*

Os recursos lado servidor são todos aqueles que serão executados ou processados em uma máquina servidora.

2.3.2.1 Linguagem de Programação: Java 5.0

Java é uma linguagem de programação orientada a objetos desenvolvida por uma pequena equipe da *Sun Microsystems*. Foi inicialmente idealizada para ser a linguagem base para projetos de *software* de produtos eletrônicos, e teve seu ápice no ano de 1995, com o sucesso mundial da *World Wide Web*.

Java é uma linguagem de alto nível[7], com sintaxe similar à do C++, e com diversas características herdadas de outras linguagens, como *Smalltalk* e *Modula*-3. É antes de tudo uma linguagem simples, fortemente tipada[8], independente de arquitetura de hardware, robusta, segura, extensível, bem estruturada, distribuída[9], *multithread*[10] e com *garbage collection*[11].

Uma das características do *Java* que o tornou ideal para uso na elaboração de aplicativos distribuídos foi a sua independência de arquitetura, pois seu compilador não gera instruções específicas para uma arquitetura computacional, mas sim instruções em código intermediário (*bytecode*), que deverão ser interpretadas por uma máquina virtual. Batizada de *JVM (Java Virtual Machine)*, a máquina virtual *Java* atua como um emulador de processador, capaz de processar os *bytecodes* e convertê-los em linguagem máquina. Desta forma, a *Sun*[12]

---

[7] Linguagens de programação de alto nível são dialetos de programação com um nível de abstração relativamente elevado, ou seja, longe do código de máquina e mais próximo à linguagem humana.

[8] Característica de uma linguagem de programação na qual cada variável deve ser declarada antes de ser utilizada, especificando-se exatamente o único tipo de dado que ela poderá conter.

[9] Característica de uma linguagem de programação que garante amplo suporte a aplicações em rede.

[10] Característica de uma linguagem de programação na qual um processo é capaz de dividir-se em duas ou mais tarefas que podem ser executadas simultaneamente.

[11] *Garbage collection* é um tipo de gerenciamento automático de memória responsável por desalocar o espaço de memória reservado aos objetos que não serão mais acessados durante o tempo de vida da aplicação.

[12] *Sun Microsystems* é uma empresa fabricante de computadores, semicondutores e *software* com sede em Santa Clara, Califórnia, no Vale do Silício. Os produtos da *Sun* incluem servidores e estações de trabalho baseados no seu próprio processador *SPARC* e no processador *Opteron*, da

compromete-se apenas em programar *JVMs* compatíveis com cada sistema operacional de mercado, eximindo os programadores de preocuparem-se com a portabilidade de seus aplicativos.

Aliado a escolha da plataforma de desenvolvimento, foram levados em consideração os princípios da facilidade de manutenção, portabilidade, rastreabilidade, reusabilidade e modularidade.

Para atender a esses princípios foram utilizados no percurso de desenvolvimento do projeto um amplo ferramental e diversos d*esign patterns*.

2.3.2.2 Uma alternativa ao CGI: Servlets

A plataforma *Sun's Java Servlet* surgiu para solucionar as duas principais falhas dos programas *CGI*[13]*.* Primeiro, os *servlets* oferecem melhor desempenho e utilização de recursos do que os programas *CGI* convencionais. Segundo, a característica multiplataforma da linguagem *Java* permite que os *servlets* sejam portáveis para qualquer sistema operacional que suporte a instalação de uma *JVM*.

O *servlet* funciona de forma análoga a um servidor *web* em miniatura. Ele recebe uma requisição e monta uma resposta. Porém, diferentemente de um servidor *web*, a *API*[14] dos *servlets* foi desenhada para auxiliar o desenvolvedor *Java* na criação de aplicações *web* dinâmicas.

Estruturalmente os *servlets* são classes *Java* convencionais, compiladas em *byte-codes* como qualquer outra. Eles possuem acesso a uma rica *API* de serviços *HTTP*[15].

AMD, nos sistemas operacionais *Solaris* e *Linux*, no sistema de arquivos de rede *NFS* e na plataforma *Java*.

[13] *Common Gateway Interface (CGI)* é um protocolo padrão para o *interface*amento de aplicações externas através de um *servidor web*. Ele permite ao servidor *web* passar as requisições de um cliente para uma aplicação externa (programadas em *C/C++, Fortran, PERL, TCL, Visual Basic, AppleScript* e etc.). Na seqüência, o servidor *web* retorna a saída do processamento para o cliente.

[14] *Application Programming Interface* (ou *Interface* de Programação de Aplicativos) é um conjunto de rotinas e padrões estabelecidos por um *software* para utilização de suas funcionalidades por programas aplicativos, isto é: programas que não querem envolver-se em detalhes da implementação do *software*, mas apenas utilizar seus serviços. De modo geral, a *API* é composta por uma série de funções acessíveis somente por programação, e que permitem utilizar características do *software* menos evidentes ao usuário tradicional.

[15] *HTTP (HyperText Transfer Protocol)* é um protocolo de comunicação entre computadores. Um protocolo define como determinados dados são transmitidos e como decodifica-los uma vez recebidos. Aplicativos *web* utilizam o *HTTP* para o tramite de dados entre o navegador de internet

Para que os servidores *web* sejam capazes de suportar a plataforma *servlet* um novo container foi anexado: o *servlet container*. Sendo assim, os *servlets* se conectam a este para tornarem-se acessíveis.

Cada *servlet* e capaz de declarar um padrão de *URL* qual irá atender. Quando uma requisição é realizada através de uma *URL* igual ao padrão declarado, o servidor *web* a direciona para o *servlet container* que irá invocar o *servlet* responsável por tratá-la.

Diferentemente dos programas *CGI* um novo *servlet* não é criado a cada requisição, mas sim uma nova *thread*. As *threads Java* utilizam menos recursos do que os processos servidor criados por programas *CGI*. Contudo os *servlets* não são *thread-safe*, ou seja, não garantem que dados compartilhados por mais de uma *thread* sejam acessados por uma única *thread* de cada vez. O tratamento desta situação fica a cargo do desenvolvedor.

Assim que um *servlet* é criado é possível utilizar o método *init()* para iniciar a programação.

2.3.2.3 Páginas Web Dinâmicas: Java Server Pages (JSP)

Embora os *servlets* sejam um grande passo em relação aos programas *CGI* eles não são capazes de resolver todas as dificuldades existentes no processo de desenvolvimento de aplicações *web*. Os desenvolvedores desta plataforma estão fadados a enviar as respostas do processamento de seus *servlets* para o cliente através do uso de métodos como o *println().* Mesmo com bibliotecas capazes de auxiliarem a montagem do código *HTML* de resposta, o aumento do tamanho das aplicações começou a gerar dificuldades de manutenção.

Da mesma forma, gerentes de projeto preferem dividir o trabalho de desenvolvimento de uma aplicação em grupos especializados. Enquanto engenheiros de *software* trabalham na modelagem os designers deveriam estar trabalhando na criação da *interface* com o usuário. No entanto, a utilização de

---

do cliente e o aplicativo rodando no servidor. O protocolo *HTTP* é um protocolo *stateless*, ou seja, que não conhece o estado das suas conexões, aceitando qualquer requisição de qualquer cliente e gerando sempre algum tipo de resposta. A não manutenção do estado das conexões é refletida no alto desempenho do protocolo, que é capaz de manipular um grande volume de requisições. Este é uma das razões pela qual a internet é capaz de abrigar uma escala de milhões de computadores.

servlets encoraja o acoplamento entre a lógica da aplicação e a lógica de apresentação, tornando inviável a divisão de tarefas.

Com o intuito de solucionar estes problemas a *Sun* buscou elaborar uma solução qual possibilitasse a utilização de técnicas de *scripting* e *templates*. A solução desenvolvida ficou conhecida como *Java Server Pages (JSP)*.

Para a construção de páginas JSP os desenvolvedores utilizam a clássica sintaxe *HTML* para a exibição de elementos estáticos e quando é necessária a exibição de dados dinâmicos são utilizadas técnicas de *scripting*. Estas técnicas consistem da definição de *tags* (marcações) que encapsulam uma lógica reconhecida pelo *JSP*. Existem três tipos de elementos de *scripting,* como o mostrado na tabela abaixo:

Elemento	Propósito
Expression	Código *Java*, colocado entre <%= e %>, usado para avaliar expressões e inserir o resultado na saída dos *servlets*
Scriptlets	Código *Java*, colocado entre <% e %>, frequentemente utilizado para criar conteúdos dinâmicos
Declarations	Código *Java*, colocado entre <%! e %>, usado para adicionar código no corpo dos *servlets*

Tabela 1: Elementos de *scripting*.

Para que as páginas *JSP* possam ser interpretadas pelo servidor *web* basta que o código *HTML* e os elementos de *scripting* sejam salvos em uma arquivo com a extensão *.jsp*. Quando um cliente requisita uma página *JSP* pela primeira vez o container traduz o código da página para um *servlet* que é compilado em um arquivo *.java*. Caso haja alguma alteração na página *JSP* o container imediatamente faz uma verificação de versões e re-compila a pagina *JSP*.

Sendo assim, tornou-se possível que a lógica de apresentação fosse separada da lógica da aplicação, e as atividades de desenvolvimento puderam ser direcionadas a áreas especializadas.

2.3.2.4 Apresentação de Dados Dinâmicos: JSP Tags (Tag Libraries)

Embora a utilização de *scriptlets* seja rápida, fácil e poderosa, a experiência nos ensina que escrever lógica da aplicação em páginas *JSP* dificulta a manutenção e minimiza o reuso do código. Todavia, os elementos de *scripting* são apenas uma das duas abordagens para a exibição de conteúdos dinâmicos em páginas *JSP.* A segunda engloba as *tags JSP*.

Uma única *tag JSP* pode estar ligada a dúzias de linhas de programação Java, porém o desenvolvedor só precisa saber como utilizá-las. O código de programação de cada *tag* fica escondido em um arquivo *.class*.

As *tags JSP* podem ser misturadas as *tags HTML* como se fizessem parte de uma mesma sintaxe. Para reutilizar o mesmo código em outras páginas, basta que o programador insira a respectiva *tag* na página desejada.

Caso o código por de traz de uma *tag* seja alterado as alterações passarão a valer pra todos os locais em que a *tag* estiver inserida. Através desta funcionalidade a utilização de *tags JSP* torna-se superior aos tradicionais *scriptlets*.

O agrupamento de várias *tags JSP* em um pacote é chamado de *Tag Library*. As *tag libraries* oferecem uma coleção de *tags* reusáveis que auxiliam no desenvolvimento de páginas *JSP.*

2.3.2.5 JavaBeans

JavaBeans são classes *Java* que estão de acordo com uma coleção de padrões responsáveis por torna-las mais fácil de ser utilizada por ferramentas de desenvolvimento e componentes.

Para qualificar um *javabean*, a classe deve ser concreta, pública e possuir um construtor sem argumentos. Os *javabeans* expõem seus campos internos como propriedades acessíveis a partir de métodos públicos padronizados. A partir desta padronização as classes *Java* podem realizar introspecção ou consultar outra classe para descobrir as suas propriedades.

Os acessos às propriedades pressupõem a utilização de dois tipos de métodos: os métodos assessores (usados para ver o estado de um *javabean*) e os métodos mutatórios (responsáveis pelas alterações de estado de um *javabean*).

Os métodos mutatórios utilizam em sua assinatura o prefixo *"set"* (em minúsculas) seguido do nome da propriedade. O primeiro caractere do nome da propriedade deve estar em maiúscula. O retorno do método é *void*, ou seja, nenhum. O método recebe um único parâmetro de qualquer tipo. Os métodos mutatórios são comumente conhecidos como *"setters"*.

```
public void setAltura(Double altura){}
```

Quadro 4: Exemplo de método mutatório para a propriedade Altura.

De forma similar são padronizados os métodos assessores. Os métodos assessores recebem como prefixo a palavra *"get"* (em minúsculas) seguido do nome da propriedade. A primeira letra do nome da propriedade deve ser em maiúscula. O tipo de retorno do método deve coincidir com o tipo do argumento passado no método mutatório. Comumente o método não possui nenhum tipo de parâmetro. Os métodos assessores são conhecidos com *"getters".*

```
public Double getAltura(){}
```

Quadro 5: Exemplo de método assessor para a propriedade Altura.

Se o método assessor retornar um valor lógico, ao invés de usarmos o prefixo *"get"*, devemos utilizar o prefixo *"is"* (em minúscula) seguido do nome da propriedade. O primeiro caractere do nome da propriedade deverá estar em maiúscula. O retorno do método será sempre um valor lógico – *boolean* ou *Boolean*. Métodos assessores lógicos não aceitam parâmetros.

```
public boolean isLigado(){}
```

Quadro 6: Exemplo de método assessor lógico para a propriedade Ligado.

Esta nomenclatura de métodos permite que outros componentes, utilizando a *API* de *reflection*[16] do *Java* possam inspecionar o *javabean* e descobrir quais são

[16] *Reflection* é um mecanismo que permite a um programa *Java* examinar ou fazer introspecção em suas propriedades e estrutura. Com este recurso, é possível em tempo de execução obter o

suas propriedades através da análise dos prefixos *"get"*, *"set"* e *"is"*. Uma vez descoberta as propriedades de uma classe um componente é capaz de utilizar os métodos assessores e mutatórios para consultar ou alterar o valor de uma propriedade.

Inicialmente a *Sun* criou os *javabeans* para auxiliar o desenvolvimento de *interface*s com o usuário, mas a popularidade foi tanta que o conceito foi difundido e os *javabeans* passaram a estar presentes em vários aspectos do desenvolvimento *Java,* componentes e *frameworks*[17].

2.3.2.6 Camada de Persistência: Hibernate Framework

Em aplicações orientadas a objetos que utilizam banco de dados relacionais como mecanismo de persistência sempre nos deparamos com o mesmo problema: a resolução do paradigma objeto-relacional.

Este paradigma, também conhecido como *object-relational impedance mismatch*, trata das dificuldades resultantes da convivência de dois modelos distintos: o modelo orientado a objetos e o relacional. Estas dificuldades são atribuídas à forma como cada modelo foi concebido. O primeiro, através de princípios da engenharia de *software* e o segundo através de princípios matemáticos. Esta diferença de concepção torna impossível a convivência entre ambos sem que haja algum tipo de tradução.

Esta tradução é comumente realizada de forma manual, através de uma técnica que consiste da mistura de comandos *SQL* ao código-fonte da aplicação. Esta solução prevê que cada operação encarregada de persistir objetos no banco de dados deverá ser capaz de converter programaticamente os atributos do objeto em uma instrução *SQL* de inserção. De forma análoga os resultados obtidos de instruções *SQL* de consulta deverão ser convertidos para objetos. Porém, esta técnica além de tornar o código complexo, confuso e ilegível, acaba por criar um forte acoplamento entre aplicação e seu banco de dados.

---

nome de todos os membros de uma classe - como atributos e métodos - e manipula-los da forma desejada.

[17] Um *framework* é uma aplicação reutilizável e semi-completa que pode ser especializada para produzir aplicações personalizadas [SPIELMANN, 2003].

Para automatizar a tradução entre os modelos e manter a independência da aplicação ao banco de dados foi utilizado no desenvolvimento deste projeto o *Hibernate*.

O *Hibernate* é um *ligthweight framework* de mapeamento objeto-relacional escrito na linguagem *Java*. É um *software* livre, de código-fonte aberto e distribuído sob a licença *LGPL*[18].

Sua função é estabelecer uma ponte de comunicação entre o modelo orientado a objetos e o relacional através do uso de mapeamentos. Estes são descritores XML[19] validados por um *DTD*[20], responsáveis por relacionar as classes e cada um de seus atributos com tabelas do banco de dados e suas respectivas colunas. Nestes descritores é possível utilizar recursos do modelo orientado a objetos, como a associação, herança, polimorfismo, composição e coleções, deixando a cargo do *framework* convertê-los para a abordagem equivalente no modelo relacional.

O *Hibernate* possui a sua própria linguagem de consulta chamada *HQL*[21], que é convertida para *SQLs* específicas para cada banco de dados suportado pelo *framework*. Isto possibilita que as aplicações clientes tenham certa independência do banco de dados utilizado.

Além disso, o processo de desenvolvimento utilizando o *Hibernate* ganha produtividade se comparado ao uso do *JDBC*[22] puro. Ele pode ser definido sinteticamente em cinco passos:

---

[18] Variação da licença *GPL (General Public License)* que permite o desenvolvimento de programas de código aberto que contenham módulos proprietários.

[19] *XML* (eXtensible Markup Language) é uma recomendação da *W3C* para gerar linguagens de marcação para necessidades especiais. Foi desenvolvida com a função de carregar dados e demonstrar o que cada dado é (metadados). Possui aplicação em *web services* e na troca de dados entre sistemas legados (de plataformas diferentes).

[20] O *DTD* (Definição de Tipo de Documento) é utilizado para especificar quais elementos ou atributos são permitidos dentro de um documento *XML*, e em que ordem no documento eles podem aparecer. Comumente o código *DTD* fica hospedado fora do documento *XML*, definindo este apenas uma referência para o primeiro. O *DTD* irá atuar como um validador do documento *XML*, comprovando sua integridade de dados em qualquer instante e assegurando que uma aplicação que não possui internamente uma descrição dos dados possa validá-los mesmo assim.

[21] Linguagem de consulta do *Hibernate* com sintaxe muito similar a linguagem *SQL*. Diferentemente da linguagem *SQL* a HQL é altamente orientada a objetos, compreendendo noções de associação, herança e polimorfismo. Ela permite que sejam escritas instruções de sintaxe similar ao *SQL*, mas que manipulam objetos e seus atributos ao invés de tabelas e colunas.

[22] *Java Database Connectivity* ou *JDBC* é um conjunto de classes e *interface*s (*API*) escritas em *Java* responsáveis pela conectividade e envio de instruções *SQL* para qualquer banco de dados relacional.

1. Criação de um arquivo *XML* contendo as configurações para que o *Hibernate* conecte ao banco de dados;
2. Criações das tabelas no banco de dados na quais os objetos serão persistidos;
3. Programação das classes persistentes;
4. Criação de descritores *XML* mapeando as classes e seus atributos para as tabelas e suas colunas respectivamente;
5. Programação das classes *DAO* responsáveis pelas operações de persistência dos objetos.

Seguindo estes cinco passos, o *Hibernate* através da leitura dos descritores *XML* das classes e utilizando o mecanismo de *reflection* é capaz de automaticamente manipular os atributos de uma classe através de seus métodos mutatórios e assessores, podendo assim populá-los, ou extrair dados para uma inserção no banco de dados.

Isto irá permitir que a persistência dos objetos em banco de dados relacionais ocorra de maneira transparente e para qualquer tipo de banco de dados suportado pelo *framework*, além de ser uma solução elegante para o paradigma objeto/relacional.

2.3.2.7 Camada de Controle: Struts Framework

O *Apache Struts* é um *ligthweight framework* para a criação de aplicações *Java* para *web* baseadas na arquitetura *MVC*. É um *software* livre, de código fonte aberto e distribuído sob a *Apache Software License [ASF, License]*.

A função principal do *framework* é prover uma camada controladora *MVC* formada por uma coleção de componentes programáveis que permitem aos desenvolvedores definir exatamente como aplicação deverá interagir com o usuário.

A estrutura do *framework* é baseada na implementação do *Sun's Model 2, MVC* e *Layer Pattern* [POSA,1996]. O Modelo 2 é implementado através do *servlet* controlador usado para gerenciar o fluxo entre as páginas *JSP*. O *MVC* e o *Layer*

Pattern são implementados através do uso de *ActionForwards*[23] e *ActionMappings*[24] responsáveis por manter o fluxo da aplicação fora da camada de visão.

Desta forma as páginas *JSP* irão fazer referência apenas a destinos lógicos que em tempo de execução serão substituídos por destinos reais pela camada controladora.

A Tabela 1 exibe as principais classes do *framework* e suas correspondências com as responsabilidades dos componentes *MVC* clássicos.

Classes	Descrição
ActionForward	Gestos do usuário ou seleção de uma tela.
ActionForm	Os dados para a mudança de estado.
ActionMapping	O evento de mudança de estado.
ActionServlet	A parte da camada controladora que recebe os gestos do usuário, mudanças de estado e direciona a aplicação para a visão apropriada.
Action	A parte da camada controladora responsável por interagir com o modelo e executar as alterações de estado ou realizar uma consulta e alertar o *ActionServlet* sobre qual deverá ser a próxima visão a ser selecionada.

Tabela 2: Classes principais do *Struts* e suas relações com a arquitetura *MVC*.

Juntamente com estas classes o *Struts* utiliza alguns arquivos de configuração e utilitários para apresentação de dados em tela para criar uma ponte sobre o "abismo" existente entre as camadas de controle e modelo. A Tabela 2 lista os arquivos de configuração e descreve seus respectivos papéis na arquitetura.

[23] Um *ActionForward* representa o destino qual a aplicação será direcionada como resultado do processamento de uma *Action*.
[24] O *ActionMapping* representa o mapeamento entre uma requisição e a *Action* responsável por tratá-la.

Arquivo	Propósito
ApplicationResources.properties	Armazena mensagens da aplicação que podem ser utilizadas para internacionalização (suporte a diferentes idiomas).
struts-config.xml	Armazena as configurações padrão dos objetos controladores, incluindo gestos do usuário, mudanças e consultas ao estado do modelo.

Tabela 3: Arquivos de configuração do *Struts*.

Para a formatação dos dados em tela o *framework* disponibiliza uma serie de *tag libraries*, mostradas na Tabela 3.

Descritor da *Tag Library*	Propósito
struts-html.tld	Extensão de *tags JSP* para auxiliar a montagem de *forms HTML*
struts-bean.tld	Extensão de *tags JSP* para o tratamento de *JavaBeans*.
struts-logic.tld	Extensão de *tags JSP* para auxiliar o teste de valores de propriedades

Tabela 4: *Tag Libraries* disponibilizadas pelo *Struts framework*.

Juntando todos os elementos do *framework* a divisão destes em camadas seria:

Camada de Visão	Camada de Controle	Camada de Modelo
JSP tag extensions	*ActionForwards* *ActionForm classes* *ActionMappings* *ActionServlet* *Action classes* *ActionErrors* *MessageResources*	*GenericDataSource*

Tabela 5: Componentes do *Struts Framework* agrupados em camadas.

Por determinação do *Layer design pattern*, os componentes devem interagir somente com componentes de sua coluna ou da coluna adjacente. Desta forma não é possível que componentes da camada de modelo interajam com a camada de apresentação.

Na prática, a camada controladora e de apresentação interagem através dos contextos de requisição, sessão e aplicação, providos pela plataforma *sevlet*, A camada controladora e de modelo irão interagir através da memória da aplicação.

A construção de aplicações utilizando o *Struts* perpassa pela criação dos *hyperlinks* necessários como *ActionForwards*, criação dos formulários *HTML* como *ActionForms* e a criação das operações servidor como *Actions*.

O fluxo entre os componentes é simples. Sempre que um *hyperlink* é acessado o *servlet* controlador é responsável por interceptar o gesto do usuário e identificar através dos *ActionMappings* qual a *Action* responsável pelo tratamento da requisição. A *Action* selecionada poderá comunicar-se com a camada de modelo para adquirir dados e exibi-los em um *ActionForm* ou extrair os dados deste para uma operação de persistência ou processamento. Por fim, a *Action* é obrigada a apontar, através de um *ActionForward*, qual a apresentação (geralmente uma pagina *JSP*) para qual o fluxo será delegado.

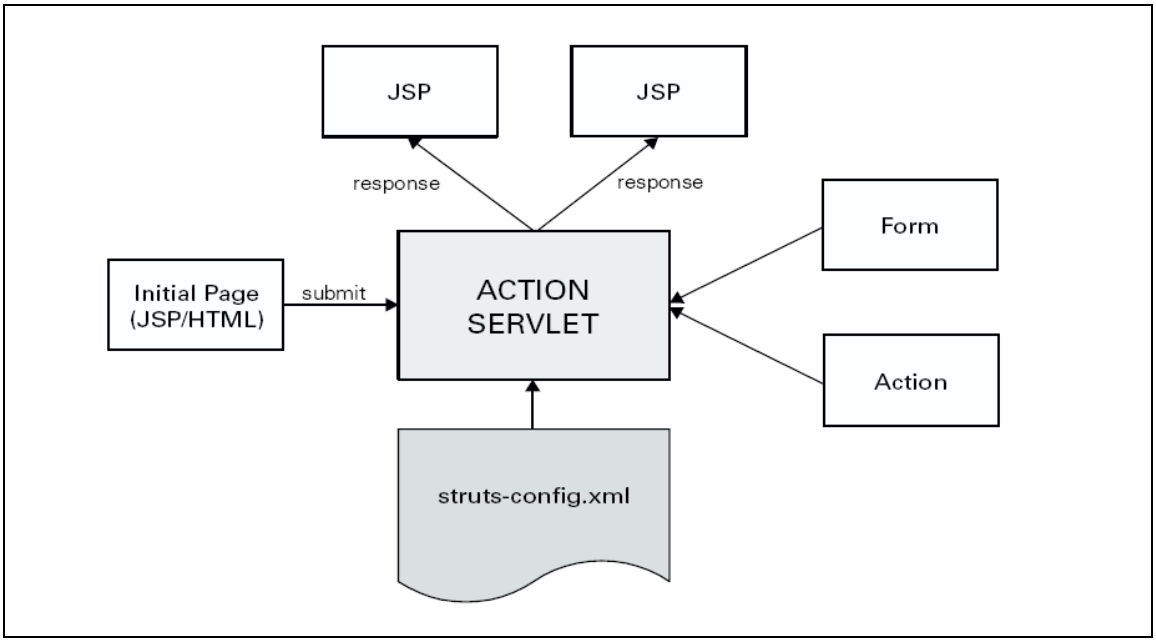


Figura 1: Esquema representando os elementos principais do *struts framework* e a comunicação entre eles.

Avaliando os desafios que o desenvolvimento *web* impõe a programadores e arquitetos de *software* o *Struts Framework* surge como uma excelente ferramenta para a criação de aplicações *web* baseadas no padrão *MVC*.

2.3.2.8 Banco de Dados: PostgreSql

O *PostgreeSQL* é um sistema gerenciador de bancos de dados objeto-relacional de grande porte. É um *software* livre distribuído sob a licença *BSD [BSD, License]*[25]. Ele é fruto do trabalho coletivo de centenas de desenvolvedores, que durante vinte anos de desenvolvimento disponibilizaram uma versão do projeto iniciado na Universidade de Berkeley, na Califórnia.

Com suporte amplo a um conjunto de recursos de nível corporativo como transações, funções, gatilhos *(triggers)* e sub-consultas *(sub-queries)*, o *PostgreeSQL* é utilizado por várias companhias e agências do governo.

[25] A licença *BSD* permite o uso e distribuição dos *softwares* sob seu registro sem custos para uso em aplicações comerciais ou não-comerciais.

2.3.2.9 Servidor de Aplicação: JBoss Server

O *JBoss* é um servidor de aplicações *Java*, de código-fonte aberto, escrito na linguagem *Java* e baseado na plataforma *Java EE*. É multiplataforma e implementa todos os serviços especificados pela arquitetura *Java EE*, possibilitando total suporte a aplicações multicamadas.. Isto permite que ele atue como uma *interface* entre os clientes, as bases de dados e os sistemas de informação corporativos .

O desenvolvimento do *JBoss* foi iniciado em março de 1999. Primeiramente concebido como um simples container *EJB*, ao longo dos anos, agregou outros componentes - como por exemplo o *web* container *tomcat*[26] – e tornou-se um dos servidores de aplicação *Java EE* mais utilizados.

2.3.2.10 IDE para Desenvolvimento Java: Eclipse

Eclipse é um *framework* de código-fonte aberto, multiplataforma, utilizado como base para a criação de aplicações *desktop*. Originalmente desenvolvido pela *IBM*, e agora sobre controle da *Eclipse Foundation*[27], o *Eclipse framework* pode ser utilizado para o desenvolvimento de qualquer tipo de aplicação *desktop*. Sua arquitetura foi concebida utilizando o conceito de *plugins* o que possibilita a extensão do ambiente de acordo com as necessidades do cliente. Tal característica permite aos fabricantes de *frameworks* – ou outras ferramentas – disponibilizarem *plugins* para a plataforma *Eclipse* com o intuito de facilitar o uso e melhorar a produtividade de suas soluções.

Geralmente o *Eclipse framework* é utilizado para a construção de *IDEs*[28], dentre as quais a mais famosa é a *Java Development Toolkit (JDT)* – a *IDE Java* desenvolvida no projeto intitulado: *The Eclipse Project*.[29]

---

[26] O *Apache Tomcat é* um container *web* desenvolvido pela *Apache Software Foundation (ASF)*. Ele implementa a especificação *servlet* e *JSP da Sun Microsystems*, provendo um ambiente de programação *Java* para *web*.

[27] A *Eclipse Foundation* é um consórcio de várias industrias de software que apostaram no *Eclipse* como o futuro *framework* para o desenvolvimento de suas *IDEs*.

[28] *Integrated Development Environment (IDE)* é um programa de computador que reúne características e ferramentas de apoio ao desenvolvimento de *software*. Sua função é permitir que os desenvolvedores obtenham um aproveitamento maior, desenvolvendo código com maior rapidez.

O *JDT* provê uma perspectiva *Java* dentro do *Eclipse* através da qual são disponibilizados vários componentes para auxílio ao desenvolvimento na plataforma *Java*.

2.3.2.11 IDE para Desenvolvimento JBoss: JBoss Eclipse IDE

A *JBoss Eclipse IDE* é um ambiente integrado de desenvolvimento desenvolvido acima do *Eclipse framework*. Com um conjunto de *plugins* ela permite aos programadores desenvolver, instalar, testar e depurar aplicações da família *JBoss* dentro da *Eclipse IDE*.

2.3.2.12 Design Patterns

Durante o desenho de aplicações devemos ao mesmo tempo ser específicos para solucionar o problema proposto e o suficientemente genéricos para integrar futuras implementações. Desejamos também, minimizar ao máximo o redesenho de partes do sistema. Arquitetos de *software* experientes sabem que nunca devemos solucionar um problema "iniciando do zero". Ao invés disso devemos procurar por soluções que possam ter nos ajudado no passado e que possam ser adaptadas e aplicadas novamente. Conseqüentemente estas soluções padronizadas poderão ser encontradas em diversas aplicações. Uma vez padronizadas estas soluções irão sempre servir para a solução de algum problema específico, possibilitando que a arquitetura orientada a objetos seja flexível, elegante e totalmente reutilizável. Esses padrões de desenho auxiliam os arquitetos menos experientes a basearem suas soluções em casos de sucesso.

Porém é muito comum que programadores e arquitetos ao encararem o desenvolvimento de novas aplicações enfrentem problemas de desenho os quais eles já tenham resolvido durante sua carreira, mas não relembram onde, como e nem quando. Para solucionar este lapso na perpetuação da experiência em arquitetura de aplicações orientadas a objetos Enrich Gamma, Richar Helm, Ralph

---

[29] *The Eclipse Project* é o principal projeto da *Eclipse Foundation*. Este inclui o desenvolvimento do próprio *Eclipse Framework*, o *Eclipse Rich Client Platform (RCP)* e o *Java Development Tools (JDT)*.

Jhonson e John Vlissides, auto-denominados *The Gang of Four*, redigiram um livro chamado *"Design Patterns – Elements of Reusable Object-Oriented Software"*, no qual documentam cada uma das soluções amplamente utilizadas no desenho da arquitetura de aplicações. A cada uma destas soluções triviais eles nomearam d*esign pattern*.

Cada d*esign pattern* é proposto de maneira que possa ser utilizado sempre que o mesmo problema de *Design* existir. No contexto de orientação a objetos, d*esign patterns* identificam classes, instâncias, seus papéis, colaborações e a distribuição de responsabilidades, reduzindo substancialmente a quantidade de entropia relacionada aos problemas que podem surgir em uma determinada arquitetura de *software*.

No que observa a documentação, os *design patterns* são altamente estruturados. Eles são documentados a partir de um modelo que identifica a informação necessária para entender o problema do *software* e a solução em termos de relacionamentos entre as classes e objetos necessários para implementação do padrão.

A *UML*[30] tem papel importante nessa documentação. É comumente usada para descrever *pattern*s e catalogá-los, incluindo diagrama de classes, de seqüência ou de interações.

Durante o desenvolvimento do SCO foram utilizados os vários *design patterns* abaixo descritos.

2.3.2.12.1 Sun's Model 2 (uma variação do clássico MVC)

Nos anos 70, quando as *interface*s gráficas *(GUI)* foram inventadas, os arquitetos de *software* enxergavam as aplicações em três grandes partes: o gerenciamento dos dados, telas e o controle das interações entre usuário e subsistemas. Logo no começo dos anos 80 o ambiente de programação

[30] *UML (Unified Modeling Language)* é uma linguagem para especificação, documentação, visualização e desenvolvimento de sistemas orientados a objetos. Sintetiza os principais métodos existentes, sendo considerada uma das linguagens mais expressivas para modelagem de sistemas orientados a objetos. Por meio de seus diagramas é possível representar sistemas de softwares sob diversas perspectivas de visualização. Facilita a comunicação de todas as pessoas envolvidas no processo de desenvolvimento de um sistema - gerentes, coordenadores, analistas, desenvolvedores - por apresentar um vocabulário de fácil entendimento.

ObjectWorks/Smalltalk introduziu este triunvirato como um *framework* de desenvolvimento para aplicações com *interface*s gráficas. Este era composto de uma tríade de componentes representado o estado da aplicação *(model)*, a apresentação em tela *(view)* e o fluxo de controle da aplicação *(controller).*

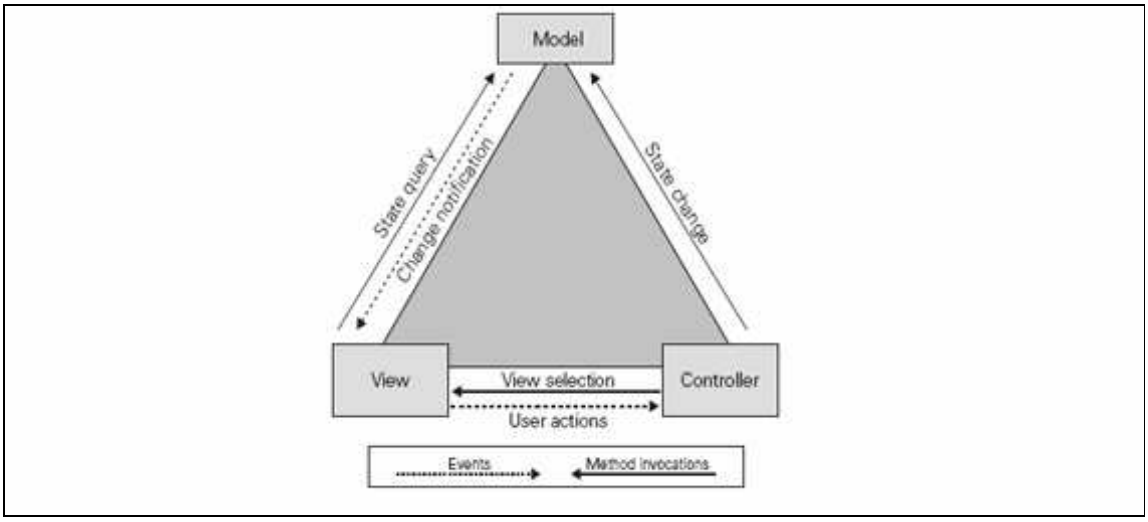


Figura 2: O *MVC* é usualmente representado como três objetos interconectados

Anos depois o *framework* Smalltalk *MVC* foi usado como estudo de caso no livro *"Design Patterns – Elements of Reusable Object-Oriented Software"* [GAMMA, 1995].

O exemplo discutido na obra comenta uma situação na qual desejamos exibir os mesmo dados, só que de formas diferentes. Ou seja, possuirmos diferentes apresentações (*views*) para um mesmo modelo (*model*). Para isso foi exaltada pelos autores a utilização do *MVC* juntamente com outro padrão: o *Observer Pattern*. Este *pattern* define que cada apresentação (view) deve estar registrada como um observador do modelo (*model*). Uma vez aplicado este padrão, a aplicação deverá manter cada uma das visões atualizadas sempre que o cliente efetuar uma mudança ou o estado do modelo for alterado. As alterações realizadas pelo cliente seriam submetidas à camada de controle (*controller*) que seria responsável por coordenar as modificações no modelo. Na seqüência, para atualizar as apresentações (*views*) o modelo enviaria uma mensagem para todos os observadores registrados.

Com o aumento da popularidade das aplicações w*eb* e o advento do *JSP*, introduzido com uma alternativa aos complexos *servlets*, foi possível o desenvolvimento de páginas w*eb* dinâmicas de forma mais rápida e intuitiva. Porém, esta nova facilidade deveria ser usada de forma responsável, e aos poucos os desenvolvedores começaram a analisar que se não tomassem o devido cuidado um projeto poderia facilmente desmoronar sobre o infindável acoplamento entre as páginas *JSP*. Ao mesmo tempo perceberam que recursos avançados de programação de *interface* só eram possíveis através do uso de complexos *scriptlets*. Estes, de difícil reuso, precisavam ser copiados em cada pagina *JSP* que o recurso fosse utilizado.

Foi neste contexto que alguns desenvolvedores voltaram seus olhos novamente para os *servlets* e perceberam que estes poderiam ser utilizados em conjunto com o *JSP*. Os *servlets* seriam responsáveis por controlar o fluxo da aplicação, atuando como um controlador, e os *JSP*s seriam encarregados de apresentar os dados em tela. Esta solução, que utiliza *servlets* e *JSPs* juntamente, ficou conhecida como Modelo 2.

Não demorou muito para que o Modelo 2 fosse comparado a clássica arquitetura *model-view-controller* e muitas discussões foram realizadas para constatar se este modelo realmente era uma implementação *MVC*. A maior controvérsia divulgada por aqueles que acreditavam que o Modelo 2 não pertencia a uma arquitetura *MVC* era o fato deste não suportar a utilização do d*esign pattern Observer*.

De fato, a razão pela qual o Modelo 2 é distinto do *MVC* clássico reside no mau funcionamento do *desing pattern Observer* em ambiente *web.* Diferentemente dos aplicativos *desktop* os aplicativos w*eb* funcionam utilizando o protocolo *HTTP*, que é essencialmente *request/response*. O cliente faz a requisição e o servidor envia uma resposta. Sem requisição não há resposta. O padrão *Observer* requer que o servidor seja capaz de enviar uma mensagem para o cliente, sem que este a tenha requisitado, informando que o estado do modelo foi alterado. Embora existam meios para implementação de tal solução, estes seriam contra a natureza do protocolo *HTTP* e com certeza encarados como um reparo mal feito.

Arquitetos de aplicações distribuídas e arquitetos de aplicações *web* estremeciam com a idéia da apresentação *(view)* ser responsável por realizar as consultas ao estado da camada de modelo.

Foi introduzido, então, no contexto *MVC* o *Layer Pattern* (padrão de camadas) [POSA,1996]. Essencialmente este padrão define que classes podem interagir apenas com classes de sua própria camada ou de uma camada adjacente. Tal abordagem impede que as dependências cresçam conforme sejam adicionados novos componentes à aplicação.

A introdução deste padrão manteve as responsabilidades principais de cada componente *MVC* intacta e ao mesmo tempo forçou a camada controladora (*controller*) a responsabilizar-se tanto pelas alterações na camada de modelo como pelas consultas ao estado desta. A camada de apresentação passou a exibir conteúdos dinâmicos a partir de dados disponibilizados unicamente pela camada controladora, ao invés de diretamente pela camada de modelo. Esta mudança permitiu que a camada de apresentação funcionasse de maneira desacoplada da camada de modelo, permitindo que a camada controladora adquira os dados a partir da camada de modelo e selecione a tela *(view)* na qual os dados serão exibidos.

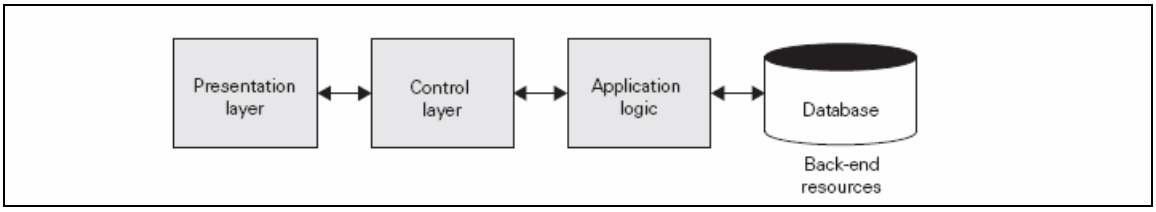


Figura 3: Camadas de uma aplicação w*eb*.

2.3.2.12.2 *Design Pattern DAO (Data Access Object)*

O *design pattern DAO* interage como uma ponte entre os componentes do sistema e a fonte de dados. Seu propósito é separar a lógica de acesso a dados da lógica de negócio, possibilitando que cada camada do sistema se concentre em sua real funcionalidade.

É muito comum nos depararmos com implementações nas quais a lógica de negócio esta misturada – portanto fortemente acoplada – a lógica de acesso a dados. *API's* para geração de conexão a banco de dados relacionais e instruções *SQL*[31], métodos para gravação de dados em arquivos texto, acessos a *Lightweight Directory Access Protocol (LDAP)*[32], acessos ao *mainframe*[33], todos misturados às regras que regem o modelo de domínio da aplicação.

Em outro contexto uma mesma aplicação pode requisitar ou persistir dados em várias fontes de dados diferentes.

Estas situações geralmente representam um desafio a arquitetura da aplicação, pois carregam um enorme potencial para criação de dependências diretas entre o código da aplicação e o código de acesso a dados. Os componentes do sistema, como objetos de domínio, *servlets* e *JSP*s precisam acessar um fonte de dados a fim de suprir suas necessidades, porém utilizando as *APIs* de uma fonte de dados e inserindo código para adquirir conectividade e acesso a dados em suas próprias implementações acabam por gerar um forte acoplamento entre estes componentes e a fonte de dados. Este acoplamento reduz a flexibilidade da aplicação e torna difícil a migração de uma fonte de dados para outra. Quando a fonte de dados for alterada, todos os componentes deverão ser alterados também, a fim de tratar a nova fonte de dados.

Desta forma, é necessário que os componentes do sistema sejam transparentes ao atual mecanismo de persistência ou fonte de dados, a fim de prover um desacoplamento dos diversos tipos de fonte de dados, facilitar futuras migrações e transformar a arquitetura do sistema em algo modular.

O *design pattern DAO* sugere que a lógica de acesso a dados seja retirada da classe de negócio e encapsulada dentro de outra classe, cuja instância é conhecida como objeto de acesso a dados (*Data Access Object* ou *DAO*). Com

---

[31] *SQL (Structured Query Language)*, ou linguagem de consulta estruturada, é uma linguagem de pesquisa declarativa para banco de dados relacionais. Devido a sua simplicidade e facilidade de uso a linguagem *SQL* tornou-se um grande padrão em banco de dados.

[32] *Lightweight Directory Access Protocol*, ou *LDAP*, é um protocolo para atualizar e pesquisar diretórios rodando sobre *TCP/IP*. Um diretório *LDAP* geralmente segue o modelo X.500, que é uma árvore de nós, cada um consistindo de um conjunto de atributos com seus respectivos valores. O *LDAP* foi criado como uma alternativa ao muito mais incômodo *Directory Access Protocol (DAP)*.

[33] O *mainframe* é um computador de grande porte, dedicado normalmente ao processamento de um volume grande de informações. Os *mainframes* são capazes de oferecer serviços de processamento a milhares de usuários através de milhares de terminais conectados diretamente ou através de uma rede.

esta divisão a classe de negócio ficará responsável unicamente por conter os métodos e atributos aliados a regra de negócio da aplicação e a classe *DAO* ficará responsável por conter os métodos para consulta e operações de *CRUD (Create, Read, Update, Delete)* do objeto de negócio.

Para gravar dados a *DAO* converte os atributos dos objetos de negócio em instruções de armazenamento e as envia para seu respectivo meio de armazenamento. Comumente em aplicações orientadas a objetos e banco de dados a *DAO* se responsabiliza por abstrair os atributos de um objeto de negócios e efetuar as devidas instruções *SQL* para as operações disponibilizadas pelo sistema.

Semelhantemente, para fazer uma consulta no banco de dados a *DAO* busca os dados do banco de dados e converte em objetos de negócio para serem manipulados pela aplicação.

Sendo assim, a *DAO* implementa o mecanismo de acesso necessário para trabalhar com cada fonte de dados e disponibiliza para os demais componentes uma *interface*[34] através da qual estes componentes irão acessá-la. Através deste artifício a *DAO* esconde completamente os detalhes da implementação de acesso a fonte de dados de seus clientes. Como a *interface* disponibilizada pela *DAO* raramente muda, mesmo que a fonte de dados seja alterada, este padrão de projeto possibilita que qualquer fonte de dados seja adaptada ao sistema sem afetar os demais componentes da aplicação. Este padrão atua, então, como um adaptador entre os componentes do sistema e as fontes de dados.

Geralmente, existe uma *DAO* para cada objeto do domínio do sistema, ou então para cada módulo, ou conjunto de entidades fortemente relacionadas.

[34] *Interface* de um objeto, na programação orientada a objetos, consiste de um conjunto de métodos que um objeto deve suportar.

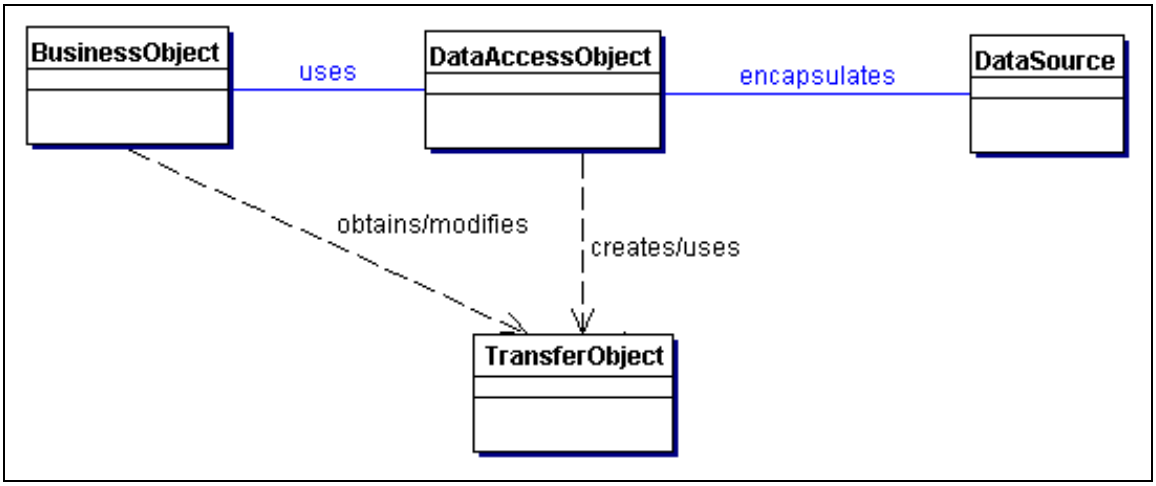


Figura 4: Diagrama de classes representando a estrutura e os relacionamentos do *Design Pattern DAO*.

2.3.2.12.3 *Design Pattern Generic DAO*

O *design pattern Generic DAO*, como o seu nome sugere, consiste da aplicação do *design pattern DAO* utilizando o recurso *generics*, disponível na linguagem *Java* a partir da plataforma versão 5.0. O *design pattern DAO* prevê que exista uma *DAO* para cada objeto do domínio do sistema, ou então para cada módulo, ou conjunto de entidades fortemente relacionadas. Porém, os métodos implementados para as operações básicas de persistência de um objeto – *CRUD (Create, Read, Update, Delete)* – repetem-se para cada objeto do domínio, para cada módulo, ou conjunto de entidades fortemente relacionadas, alterando-se apenas a tipagem do objeto de domínio que é manipulado pela *DAO*.

Sendo assim, para que não haja repetição na implementação de cada *DAO* o *design pattern Generic DAO*, faz uso de um artifício extremamente poderoso, o *generics*. Este é um recurso de programação com o qual os algoritmos são escritos em uma gramática extensível - ou seja, que pode ser herdada – em que as classes especializadas são capazes de adaptarem-se ao modelo base com a condição de especificar os tipos de variáveis a serem utilizados. Desta forma é possível parametrizar os tipos de variáveis de uma classe base, podendo ela manipular através de sua implementação qualquer classe que lhe seja fornecida como parâmetro.

A utilização do *design pattern Generic DAO* amplia a reutilização do código, o que facilita a manutenção e aumenta a produtividade durante o desenvolvimento, pois ao invés de replicarmos o mesmo código para cada *DAO* de nossa aplicação, apenas definimos uma classe genérica – de tipos parametrizados – e lançamos mão do uso da herança para as demais *DAOs*. Como condição para o uso da classe genérica o *Java*, em tempo de compilação, obriga a classe especializada a fornecer os tipos de variáveis que deverão ser utilizadas pelo algoritmo genérico.

Isto permite que as operações triviais e repetitivas de CRUD sejam programadas em uma classe genérica (independente de tipo) que será herdada por todas as demais *DAO* especializadas. Estas ao especializarem-se englobam os métodos da primeira estipulando os tipos de variáveis a serem tratados, podendo também implementar novos métodos se necessário.

2.3.2.12.4 *Design Pattern Factory*

O *design pattern factory* consiste de uma classe (*factory*) responsável por criar instâncias de classes que implementam uma mesma *interface*. Quando necessitamos de uma instância de determinada classe a *factory* nos retorna os objetos de forma dinâmica e sem que tenhamos que nos ater aos detalhes de como ele foi construído e sem especificar exatamente qual classe iremos utilizar. Isto permite que objetos diferentes, mas de classes que implementam a mesma *interface*, possam ser utilizados pelo sistema de forma transparente e desacoplada através da referência a uma *interface*.

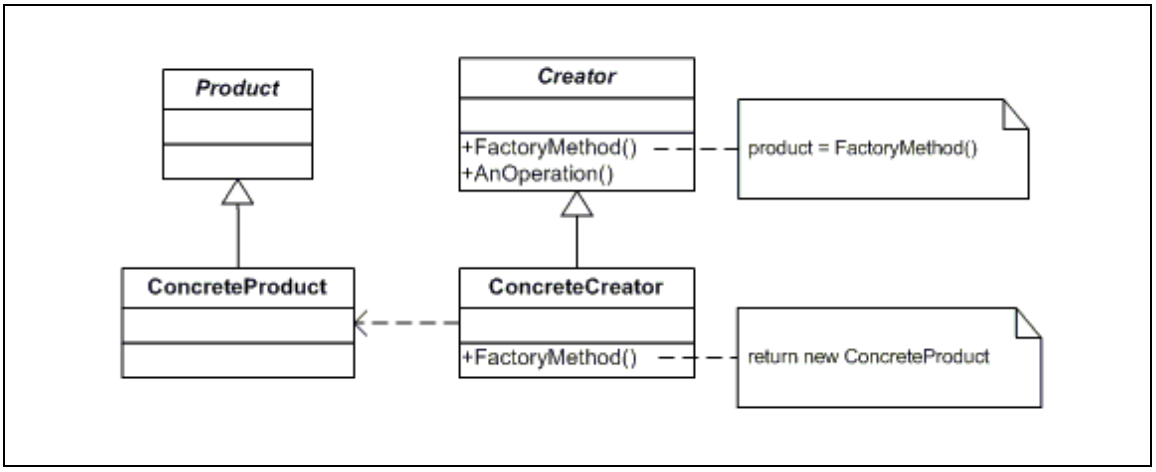


Figura 5: Diagrama de classes representando a estrutura e os relacionamentos do *Design Pattern Factory*.

2.3.2.12.5 *Design Pattern Abstract Factory*

O d*esign pattern Abstract Factory* introduz uma forma de encapsular um grupo de fábricas em uma determinada categoria. Neste padrão as fábricas são programadas como implementações concretas da fábrica abstrata. Este *pattern* está a um nível de abstração bem elevado e pode ser usado quando é necessário retornar uma das muitas classes de objetos relacionadas, cada qual podendo retornar diferentes objetos. Sendo assim, o padrão *Abstract Factory* é uma fábrica de objetos que retorna uma das várias fábricas. Isto permite que os componentes da aplicação usem a fabrica abstrata para servir o tipo de fabrica concreta necessária Os ganhos em manutenção da aplicação também são relevantes, porque basta que uma fábrica herde e implemente os métodos da fábrica abstrata para poder atuar como uma fábrica concreta da aplicação.

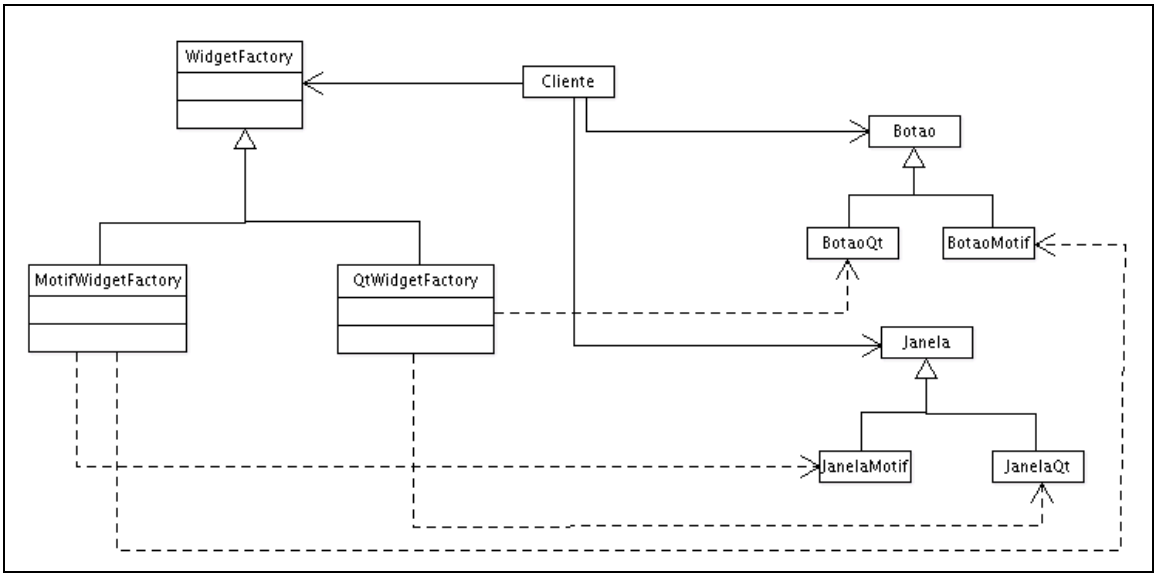


Figura 6: Diagrama de classes representando a estrutura e os relacionamentos do *Design Pattern Abstract* Factory.

2.3.2.12.6 *Design Pattern Façade*

O d*esign pattern Façade* implementa uma *interface* unificada para um conjunto de *interface*s em um subsistema e define uma *interface* de alto nível para facilitar o uso deste subsistema. Em aplicações distribuídas a realização de um caso de uso que envolve a invocação de diversas classes do modelo de domínio resulta em um alto índice de chamadas remotas trazendo prejuízos ao desempenho da aplicação.

Para resolver esse problema o *design pattern Façade* é utilizado para ocultar a alta interatividade entre as classes do modelo de domínio, através de *interface*s locais, atrás de um único método remoto. Isto resulta em menos invocações remotas, maior simplicidade na *interface* de serviços para o cliente (disponibilizando um ponto central para a realização de um caso de uso), e contribuindo para o desacoplamento entre as camadas, uma vez que todos os relacionamentos e dependências entre as entidades envolvidas na realização do caso de uso ficam transparentes ao cliente.

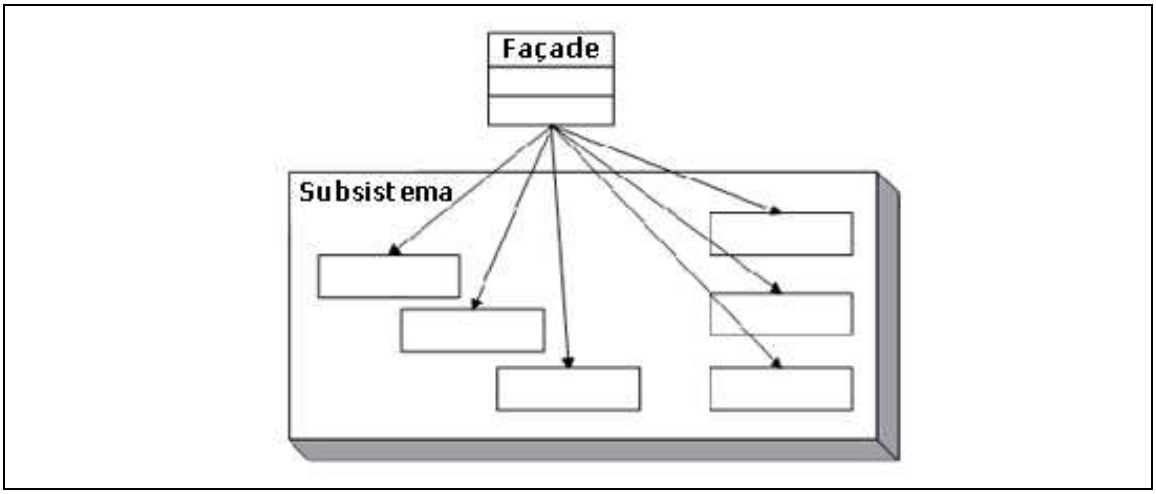


Figura 7: Diagrama de classes representando a estrutura e os relacionamentos do *Design Pattern Façade*.

2.3.2.12.7 *Design Pattern Singleton*

O *design pattern Singleton* aplica-se quando desejamos que determinada classe possua apenas uma instância de fácil acesso. Por exemplo: podemos ter várias instâncias de impressora em uma aplicação, porém para gerenciar uma fila de impressão necessitamos apenas de uma instância da classe fila de impressão. Utilizando o *singleton* é possível garantir que determinada classe terá sempre uma única instância. Para isso este *pattern* define que a própria classe deverá ser capaz rastrear esta instância única - interceptando as requisições para a criação de novos objetos - garantindo aos seus clientes que nenhuma outra instância será criada.

A estratégia para aplicação deste *pattern* concentra-se em tornar o construtor da classe privado[35], criar um atributo privado do mesmo tipo da classe e implementar um método responsável por utilizar este construtor restrito para instanciar um objeto, armazená-lo no atributo privado e retornar este. O método irá verificar se o atributo privado não está nulo – ou seja, se uma instância já foi armazenada - caso esteja, ele apenas a retorna. Caso este atributo privado seja nulo o método cria uma instância, armazena-a no atributo e a retorna.

[35] O modificador de acesso *private* (privado) torna o acesso a métodos e atributos disponíveis apenas dentro da classe. Objetos externos não são capazes de acessar métodos ou atributos privados de outra classe.

Portanto, centralizar a instanciação de objetos neste método torna possível garantir a unicidade de instâncias de uma classe.

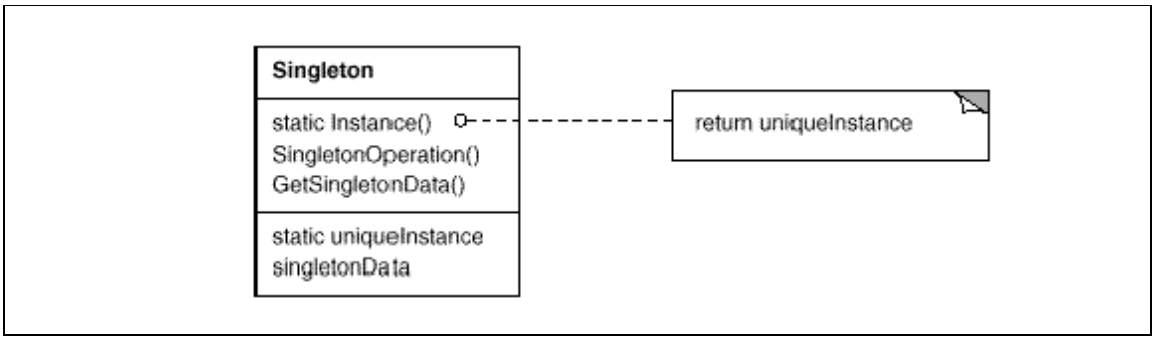


Figura 8: Diagrama de classes representando a estrutura e os relacionamentos do *Design Pattern Singleton*.

2.3.2.12.8 *Design Pattern Open Session in View (session-per-request pattern)*

O d*esign pattern Open Session in View* é uma implementação do session-per-request *pattern* e seu intuito é descrever uma solução para o tratamento de sessões e transações na utilização do *Hibernate*.

Para cada agrupamento de operações de acesso a dados o *Hibernate* define um segmento chamado unidade de trabalho *(work unit)*. As unidades de trabalho são usualmente conhecidas como sessões do *Hibernate*, pois o escopo de uma unidade de trabalho é exatamente o mesmo de uma sessão. Cada sessão funciona como uma camada responsável por gerenciar as entidades persistentes[36] (*persitent*), realizar consultas e executar comandos *SQL* gerados pela *API* do *Hibernate*.

Além da sessão o *Hibernate* também utiliza o conceito de transação. Comandos *SQL* devem sempre estar contidos em uma transação, para que seja possível realizar o *commit*[37] e o *rollback*[38] das operações, garantindo maior consistência e integridade dos dados. Porém, para utilizar este recurso é necessário que os marcos de delimitação de uma transação sejam bem definidos

[36] Objetos persistentes são os objetos que já foram persistidos no banco de dados e que ainda fazem parte de uma sessão do *Hibernate*.
[37] Operação na qual são enviados definitivamente para o banco de dados às mudanças executadas pelas operações.
[38] Operação na qual as mudanças existentes desde o último *commit* ou *rollback* são descartadas.

na implementação, de forma explicita no código, ou através de mecanismos de programação.

A primeira vista é extremamente convidativo implementar as demarcações de transação e de sessão explicitamente em cada um dos métodos das *DAOs* da aplicação. Porém não cabe a elas a responsabilidade de conhecerem como uma sessão é adquirida ou como uma transação inicia ou termina. Aplicar as demarcações de transação e de sessão nas *DAOs* consiste em erro grave na arquitetura da aplicação, pois além de estarmos violando a responsabilidade de um componente, estaríamos utilizado um *anti-pattern*, o *session-per-operation*[39].

Outro problema encontrado em aplicações w*eb* que utilizam o *Hibernate* é a tentativa de acesso a objetos da sessão quando esta já foi finalizada. Ao programarmos as fronteiras de sessão e transação em um *servlet* as operações executadas para uma consulta são respectivamente: abertura da sessão, inicio da transação, operações de acesso a dados, *commit* da transação e encerramento da sessão. Após isso o controle é passado a camada de visualização (*JSP*) que deverá exibir os dados em tela. Porém o *Hibernate* possui um recurso de carregamento tardio de coleções *(lazy loading)*, no qual os dados das coleções são recuperados do banco somente no momento do uso. Pode ocorrer que a página *JSP* queira exibir os dados de uma coleção de forma tabular, e quando a operação de carga da dos dados da coleção for executada não haverá nenhuma sessão, tão logo nenhuma transação. Será gerado um erro e a operação abortada. Isto ocorre porque as demarcações de sessão e transação não seguem a lógica de *request/response* (requisição/resposta) contemplada nas aplicações w*eb*.

A solução para este impasse é manter a sessão do *Hibernate* aberta até que a *view* (página *JSP*) seja inteiramente processada. É isto que o d*esign pattern Open Session in View* propõe. Ele descreve uma solução na qual um interceptador irá capturar a requisição *HTTP*, abrir uma nova sessão, iniciar uma transação, transferir o processamento para o *servlet* responsável por executar a lógica de programação, aguardar a resposta ser enviada para o cliente e após todo o trabalho ser realizado, executar o *commit* da transação e o fechamento da sessão.

[39] *Anti-pattern* no qual cada método que realiza alguma operação no banco de dados dispõe de sua própria sessão e transação. São raríssimas as situações em que este *anti-pattern* pode ser empregado com sucesso.

Dentro do servidor de aplicação, mais especificamente no container *Servlet* um componente que atua como um ótimo interceptador é o *Servlet*Filter. Ele é executado sempre antes que uma requisição ou uma resposta é enviada para o cliente.

Sendo assim, a implementação deste *pattern* dentro do componente *ServletFilter* possibilita aos desenvolvedores w*eb* criarem um mecanismo padronizado para o gerenciamento de sessões e transações, contribuindo assim para a criação de aplicativos robustos utilizando o *Hibernate*.

3 PROCESSO DE DESENVOLVIMENTO DO SCO

O processo de desenvolvimento do SCO foi iniciado com a definição da arquitetura da aplicação, que depois de consolidada permitiu que cada uma de suas unidades construtivas – as camadas da aplicação – fossem implementadas separadamente.

3.1 Definição da Arquitetura da Aplicação

A definição da arquitetura do SCO foi realizada através da análise e definição dos seguintes elementos:

- A organização da aplicação.
- Elementos estruturais, suas interfaces, suas colaborações e composições.
- Funcionalidade, desempenho, reuso, entendimento, restrições econômicas e tecnológicas.
- A composição de elementos estruturais e comportamentais em subsistemas ou camadas progressivamente maiores.
- O estilo arquitetural optado, no caso o *Sun´s Model 2*, uma variação do clássico *MVC*.

Desta forma, as camadas da aplicação foram divididas conforme a figura abaixo.

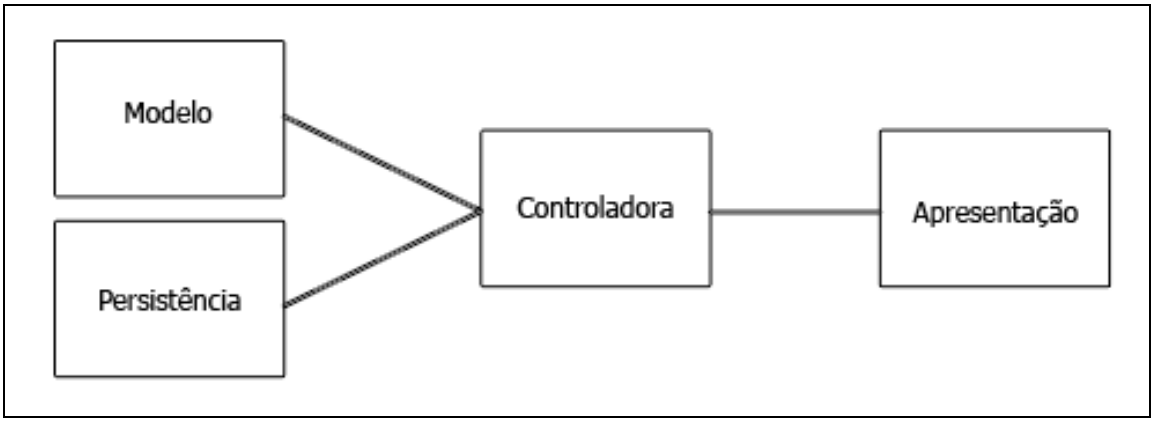


Figura 9: Divisão de camadas do SCO.

Nas seções a seguir serão detalhadas quais as funções, a justificativa e a modelagem de cada uma das camadas. Serão abordados também quais as metodologias utilizadas, o uso destas metodologias e os benefícios gerados por estas em cada camada da aplicação.

3.2 A Camada de Persistência

A camada de persistência é responsável por encapsular toda a lógica necessária para que os objetos da camada de modelo sejam gravados ou recuperados do banco de dados *PostgreSql*. Ela utiliza o *Hibernate Framework* como mecanismo de persistência objeto-relacional e para reduzir o acoplamento ao tipo de banco de dados. Para garantir a modularidade, a facilidade de reutilização e prover um ponto único de acesso a seus serviços, a camada de persistência implementa os *design patterns DAO*, *Generic DAO*, *Factory* e *Abstract Factory*.

3.2.1 Por que Criar Uma Camada de Persistência?

Primeiramente para justificar a criação de uma camada de persistência é necessário elucidar o que são objetos de negócio e quais são suas responsabilidades.

Objetos de negócio representam entidades de uma aplicação às quais o usuário cria, acessa e manipula durante a execução de um caso de uso. Eles formam o núcleo de qualquer aplicação e são responsáveis por armazenar os dados e modelar o comportamento do negócio.

Um sistema geralmente possuiu vários objetos de negócio que interagem para disponibilizar as funcionalidades. Uma vez que um processo de negócio tenha sido finalizado os objetos de negócio tendem a ser persistidos (gravados) em um meio de armazenamento permanente para uso posterior.

Tipicamente a responsabilidade de persistência dos dados de um objeto é erroneamente atribuída ao objeto de negócio. Nesta perspectiva os métodos para acesso a dados são programados juntamente aos objetos de negócio, que tem

sua responsabilidade violada, pois além de armazenarem dados e modelar o comportamento da aplicação passam também a serem responsáveis por gerenciar suas operações de acesso a dados.

Esta abordagem é extremamente frágil. Imaginemos que o meio de armazenamento utilizado seja o banco de dados *PostgreSql* e que tenhamos programado os métodos de acesso a dados nos objetos de negócio, utilizando a *API JDBC* e a linguagem *SQL*. Logo teríamos a lógica de acesso a dados, em linguagem *SQL*, misturada à lógica do negócio, em linguagem *Java*, criando um forte acoplamento entre as duas. Caso fosse necessário alterar o tipo de banco de dados ou até mesmo alterar o meio de armazenamento, certamente haveria um enorme impacto na manutenção da aplicação, que além de ter partes reescritas, precisaria ser totalmente recompilada. Isto torna claro que os objetos de negócio não devem ser responsáveis por gerenciar acesso a dados e muito menos devem conhecer como estas operações são realizadas.

Para a solução deste acoplamento é necessário encapsular a lógica de acesso a dados em uma camada responsável por fornecer este serviço: a camada de persistência.

3.2.2 A Modelagem

A modelagem da camada de persistência foi realizada de acordo com o diagrama de classes abaixo:

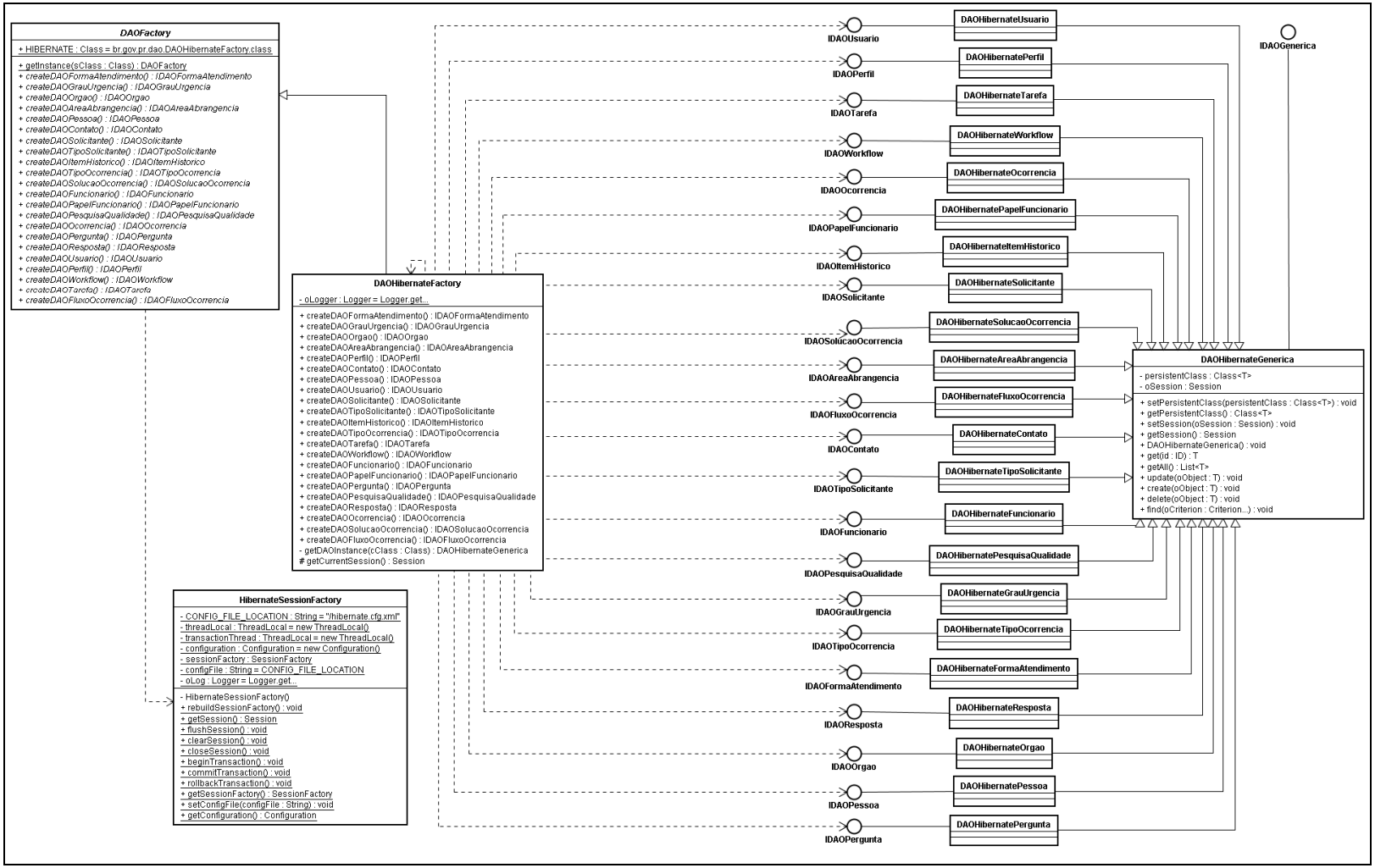


Figura 10: Diagrama de classes da camada de persistência.

3.2.3 A Implementação

O desenvolvimento da camada de persistência foi realizado através de uma seqüência de operações cujo objetivo era priorizar a facilidade de manutenção, favorecer a reutilização de código e garantir a modularidade da camada.

Cada uma destas operações teve como resultado uma melhoria para atingir os objetivos descritos, porém ao mesmo tempo ofereceram novos desafios de arquitetura que depois de refinados e solucionados permitiram a criação de uma camada de persistência robusta.

Estas operações foram respectivamente o desacoplamento da lógica de negócio da lógica de acesso a dados, o desacoplamento da lógica de acesso a dados do tipo de banco de dados, o desacoplamento da aplicação do meio de armazenamento, a generalização das operações básicas realizadas pelos objetos de acesso a dados e a formação de uma entidade única para acesso aos serviços de persistência.

3.2.3.1 Desacoplando a Lógica de Negócio da Lógica de Acesso a Dados

O desacoplamento entre a lógica de negócio e a lógica de acesso a dados foi realizada através do uso de uma técnica de sucesso e altamente explorada: o *design pattern DAO*. Este padrão de desenho documenta uma solução padronizada para a separação da lógica de negócio da lógica de acesso a dados, preservando as responsabilidades naturais dos componentes envolvidos. A solução sugere que a lógica de acesso a dados seja encapsulada em objetos de acesso a dados chamados *DAO (Data Access Object)*. Cada *DAO* é responsável por implementar os métodos de acesso a dados de um objeto de negócio ou conjunto de objetos de negócio, para um tipo específico de meio de armazenamento (banco de dados, *mainframe*, arquivos texto, *XML* e etc).

Foram criadas *DAOs* para todos os objetos de negócio da aplicação que precisassem efetuar operações de consulta ou *CRUD* no banco de dados *PostgreSql*.

Esta separação de responsabilidades proporcionou que os objetos de negócio e os objetos de acesso a dados pudessem funcionar de forma independente e quando um objeto de negócio precisa realizar uma operação de acesso a dados ele simplesmente utiliza a *DAO* programada para suportá-lo.

3.2.3.2 Desacoplando a Lógica de Acesso a Dados do Tipo de Banco de Dados

A utilização do *design pattern DAO* trouxe grandes resultados para a modularidade da aplicação no que compete à separação entre a camada de persistência e a camada de modelo. Porém os objetos de acesso a dados, se programados utilizando puramente a *API JDBC* com instruções *SQL* ainda revelam um forte acoplamento com o banco de dados. Sempre que houvesse uma alteração no esquema do banco de dados seria necessária a alteração de várias instruções *SQL* encapsuladas nas *DAOs*, o que certamente iria refletir em uma recompilação da aplicação.

Outro problema na utilização desta abordagem é a sua eficiência na resolução do paradigma objeto-relacional, pois codificar manualmente as operações de acesso a dados para cada objeto de negócio consome grande parte do tempo de desenvolvimento e dificulta futuras manutenções.

Portanto, para reduzir o acoplamento com o banco de dados e fornecer uma solução robusta para o paradigma objeto relaciona foi utilizado o *Hibernate Framework*.

O *Hibernate* é um *framework* de mapeamento objeto-relacional *(ORM – Object Relational Mappping)*. Sua função é automatizar o processo de persistência de objetos em bancos de dados relacionais através do uso de metadados que descrevem o mapeamento entre os objetos persistentes e suas propriedades com as tabelas e colunas do banco de dados.

Este mapeamento é feito através do uso de descritores *XML* (arquivos *XML* estruturados por um *DTD*) nos quais é possível relacionar classes e suas propriedades com as tabelas e respectivas colunas. Nestes descritores, além de realizar os mapeamentos, é possível utilizar recursos do modelo orientado a objetos, como a associação, herança, polimorfismo, composição e coleções deixando a cargo do *framework* traduzi-los para a abordagem equivalente do modelo relacional.

Uma vez compostos os mapeamentos, o *Hibernate* realiza a leitura dos descritores *XML* das classes e utilizando o mecanismo de *reflection* é capaz de automaticamente manipular os atributos destas através de seus métodos mutatórios e assessores. Isto só é possível porque as classes que irão usufruir dos serviços do *framework* precisam obrigatoriamente implementar o padrão *Javabean*. Sob esta condição o *framework* é capaz de automaticamente popular um objeto com dados retornados do banco de dados ou extrair os dados dos atributos de um objeto para composição automática de instruções *SQL* para os diferentes dialetos dos bancos de dados de mercado suportados pelo *framework*. Esta característica permite que a aplicação desfrute de independência do tipo de banco de dados relacional utilizado, pois existindo a necessidade da alteração do tipo de banco de dados o impacto da alteração no código da camada de persistência é nulo.

A *API* do *Hibernate* oferece componentes para a execução das operações básicas de *CRUD* dos objetos persistentes e uma linguagem – a *HQL (Hibernate Query Language)* – de sintaxe similar a *SQL*, só que altamente orientada a objetos, que permite a manipulação de objetos e seus atributos ao invés de tabelas e colunas. Esta rica *API* elimina a necessidade do uso da linguagem *SQL* para implementação dos métodos de acesso a dados, pois oferece uma abstração total do modelo relacional e centraliza o funcionamento da camada de persistência acima da *API* do *Hibernate* e a capacidade desta em manipular os descritores *XML* para a execução dos métodos de acesso a dados.

Portanto, a centralização das operações de acesso a dados no *Hibernate*, torna o código da camada de persistência livre do uso de instruções *SQL*

estáticas. As instruções *SQL* não deixam de existir, porém são geradas pelo *framework* como um produto final da tradução entre o modelo orientado a objetos e o modelo relacional. E como as instruções são compostas dinamicamente pelo *framework* através da leitura dos descritores *XML* dos objetos persistentes, qualquer alteração no esquema do banco de dados irá refletir apenas nestes descritores, que se alterados não precisam ser recompilados. E por fim, se houver a necessidade de alterar o tipo de banco de dados utilizado o *Hibernate* via parâmetros de configuração é capaz de gerar instruções *SQL* para o dialeto de todos os tipos de bancos de dados do mercado.

3.2.3.3 Desacoplando a Aplicação do Meio de Armazenamento

A utilização do *Hibernate* como *framework* de persistência objeto relacional permitiu que a aplicação atuasse de forma independente do tipo de banco de dados relacional utilizado, porém a implementação da camada de persistência é fortemente acoplada a este *framework*, logo fortemente acoplada ao tipo de meio de armazenamento com o qual o *Hibernate* trabalha, os bancos de dados relacionais. Um objeto de negócio da camada de modelo que seja cliente da camada de persistência obtém apenas serviços especializados para o acesso a dados em bancos de dados relacionais.

Assumir que a camada de persistência ofereça serviços apenas para este tipo de meio de armazenamento é a solução comumente adotada por vários arquitetos de *software*, porém não é possível afirmar com certeza que a aplicação sempre utilizará bancos de dados relacionais como meio de armazenamento. Mas podemos afirmar com maior certeza que uma mudança drástica em determinada política de uma companhia poderia refletir, por exemplo, na troca do meio de armazenamento da aplicação. Uma aplicação antes utilizando bancos de dados relacionais pode, por exemplo, passar adotar a persistência de dados em arquivos *XML* ou até mesmo em um *mainframe*. E para estas situações a aplicação deve estar arquitetada de forma a suportar a mudança, com o mínimo de manutenção necessária.

Para garantir que a aplicação suporte a utilização de outros meios de armazenamento foram utilizados os *design patterns Factory* e *Abstract Factory*.

Comumente uma classe *Java* é instanciada através do comando *new* seguido do nome da classe a ser instanciada, o que cria uma referência explicita em código ao objeto concreto a ser criado.

```
Orgao oOrgao = new Orgao();
```

Quadro 7: Exemplo de uma classe sendo instanciado em Java.

Partindo desta abordagem clássica um objeto de negócio que desejasse fazer uso da camada de persistência precisaria instanciar explicitamente a classe *DAO* concreta a qual fosse utilizar. Logicamente, esta abordagem estaria por acoplar os clientes à estrutura da camada de persistência.

Para resolver a situação descrita acima foi criada uma *DAO Factory* (fábrica de *DAOs*) na camada de persistência. Uma fábrica consiste de uma classe *Java* com métodos responsáveis por instanciar objetos que implementem uma mesma *interface*. Quando um objeto é necessário o cliente não precisa reportar-se diretamente a classe deste, mas sim solicita á fábrica que retorne uma *interface* para este objeto, e não o objeto concreto. Isto permite que os objetos sejam criados de forma dinâmica e sem que os clientes da fábrica tenham conhecimento de como, ou qual, objeto concreto foi disponibilizado para o uso.

Esta característica é possível, pois a fábrica utiliza *interfaces*. Uma *interface* é um artefato de programação que define um conjunto de assinaturas de métodos que uma classe deve suportar (implementar). Uma classe que implemente determinada *interface* é obrigado a oferecer a implementação dos métodos definidos por ela. Quando uma classe realiza esta operação dizemos que ela esta de acordo com o “contrato” firmado pela *interface*. Isto permite que classes diferentes, mas que implementam uma mesma *interface* possam ser utilizadas ou manipulados de uma mesma forma, pois possuem características em comum.

A utilização de *interface*s é muito comum para esconder a implementação de componentes ou camadas de um *software*. Um sistema operacional, por exemplo, oferece *APIs* (*interface*s de programação de aplicativos) através dos

quais um programa externo possa utilizar recursos do sistema sem que os detalhes da implementação sejam revelados ao programador, isolando o sistema operacional de eventuais erros cometidos pelo programa. Da mesma forma, componentes de *software* utilizam *interface*s padronizadas para criar uma camada de abstração que facilite a reutilização e manutenção da aplicação. Neste cenário a *interface* de uma camada "A" deve ser mantida em separado da sua implementação e qualquer outra camada "B", que interaja com "A", deve ser forçada a fazê-lo apenas através da *interface*. Este mecanismo permite que uma alteração em "A", não influencie em "B", desde que a utilização de "A" por "B" satisfaça o que foi "acordado" na *interface*.

Para criação da fábrica de *DAOs* foi necessário, primeiramente, criar uma interface para cada *DAO* de objeto de negócio da aplicação. Estas interfaces serviram para definir quais os métodos que cada *DAO* concreta é obrigada a implementar. Isto permitiu que qualquer classe que "assine o contrato da interface" (implemente a interface) possa atuar como uma *DAO* para aquele objeto de negócio específico.

Para servir nossos propósitos, podemos criar *DAOs* com implementações para diferentes meios de armazenamento e garantir a compatibilidade destas com a aplicação fazendo com que elas implementem uma mesma interface. A interface *DAO* carrega a informação do que a *DAO* concreta pode fazer e a ultima é guardiã do como fazer.

Criadas as interfaces para cada *DAO* e as *DAOs* concretas para os mais variados meios de armazenamento é necessário que algum componente da camada de persistência seja o responsável por gerir qual *DAO* concreta deverá ser utilizada em cada caso. Surge então a aplicação da fábrica de *DAOs*.

Esta fábrica foi construída de forma a oferecer métodos para a criação de *DAOs* dos objetos de negócio para a persistência de dados utilizando o *Hibernate*, ou seja, uma "*Hibernate DAO Factory*" (uma fábrica de *DAOs* exclusiva para o *Hibernate*). Ela é responsável por implementar métodos que retornem a interface para uma *DAO* concreta. Esta *DAO* concreta pode ser escolhida pelo método com base em parâmetros que ele tenha recebido, ou o método pode não possuir

nenhuma lógica e simplesmente retornar uma instância da *DAO* concreta. O que é importante perceber é que foi definido um local central responsável pela forma como as *DAOs* da camada de persistência são criadas. Neste local podem ser implementadas os mais variados tipos de funções, como o rastreamento da criação de *DAOs* através de *logs,* implementação de segurança e tudo mais que esteja ao alcance da imaginação do programador. Outro aspecto importante é que na criação de uma *DAO*, não é mais necessário à utilização do comando *new*, pois a fábrica já retorna uma instancia do objeto concreto como uma interface. Caso fosse necessária uma manutenção no nome das *DAOs* ou até mesmo a substituição de uma *DAO* por outra, a alteração na fábrica faria valer para todo o restante da aplicação, diminuindo consideravelmente o esforço de manutenção.

```
//Imports suprimidos

public class DAOHibernateFactory extends DAOAbstractFactory{

	private static Logger oLogger =
Logger.getLogger(DAOHibernateFactory.class.getName());

	public IDAOFormaAtendimento createDAOFormaAtendimento() throws
DAOConcreteFactoryException {
		oLogger.info("CRIOU a DAOHibernateFormaAtendimento");
		return
(IDAOFormaAtendimento)getDAOInstance(DAOHibernateFormaAtendimento.class);
	}

	public IDAOGrauUrgencia createDAOGrauUrgencia() throws
DAOConcreteFactoryException {
		oLogger.info("CRIOU a DAOHibernateGrauUrgencia");
		return
(IDAOGrauUrgencia)getDAOInstance(DAOHibernateGrauUrgencia.class);
	}

	public IDAOOrgao createDAOOrgao() throws
DAOConcreteFactoryException{
		oLogger.info("CRIOU a DAOHibernateOrgao");
		return (IDAOOrgao)getDAOInstance(DAOHibernateOrgao.class);
	}

	public IDAOAreaAbrangencia createDAOAreaAbrangencia() throws
DAOConcreteFactoryException{
		oLogger.info("CRIOU a DAOHibernateAreaAbrangencia");
		return
(IDAOAreaAbrangencia)getDAOInstance(DAOHibernateAreaAbrangencia.class);
	}
```

Quadro 8: Implementação da fábrica de *DAOs* do *Hibernate (Hibernate DAO Factory)*

```
public IDAOPerfil createDAOPerfil() throws DAOConcreteFactoryException{
            oLogger.info("CRIOU a DAOHibernatePerfil");
            return (IDAOPerfil)getDAOInstance(DAOHibernatePerfil.class);
      }

      public IDAOContato createDAOContato() throws
DAOConcreteFactoryException{
            oLogger.info("CRIOU a DAOHibernateContato");
            return
(IDAOContato)getDAOInstance(DAOHibernateContato.class);
      }

      public IDAOPessoa createDAOPessoa() throws
DAOConcreteFactoryException{
            oLogger.info("CRIOU a DAOHibernatePessoa");
            return (IDAOPessoa)getDAOInstance(DAOHibernatePessoa.class);
      }

      public IDAOUsuario createDAOUsuario() throws
DAOConcreteFactoryException{
            oLogger.info("CRIOU a DAOHibernateUsuario");
            return
(IDAOUsuario)getDAOInstance(DAOHibernateUsuario.class);
      }

      public IDAOSolicitante createDAOSolicitante() throws
DAOConcreteFactoryException{
            oLogger.info("CRIOU a DAOHibernateSolicitante");
            return
(IDAOSolicitante)getDAOInstance(DAOHibernateSolicitante.class);
      }

      public IDAOTipoSolicitante createDAOTipoSolicitante() throws
DAOConcreteFactoryException{
            oLogger.info("CRIOU a DAOHibernateTipoSolicitante");
            return
(IDAOTipoSolicitante)getDAOInstance(DAOHibernateTipoSolicitante.class);
      }

      public IDAOItemHistorico createDAOItemHistorico() throws
DAOConcreteFactoryException{
            oLogger.info("CRIOU a DAOHibernateItemHistorico");
            return
(IDAOItemHistorico)getDAOInstance(DAOHibernateItemHistorico.class);
      }

      public IDAOTipoOcorrencia createDAOTipoOcorrencia() throws
DAOConcreteFactoryException{
            oLogger.info("CRIOU a DAOHibernateTipoOcorrencia");
            return
(IDAOTipoOcorrencia)getDAOInstance(DAOHibernateTipoOcorrencia.class);
      }
```

Quadro 9: Implementação da fábrica de *DAOs* do *Hibernate (Hibernate DAO Factory)* (Continuação)

```
	public IDAOTarefa createDAOTarefa() throws
DAOConcreteFactoryException{
		oLogger.info("CRIOU a DAOHibernateTarefa");
		return (IDAOTarefa)getDAOInstance(DAOHibernateTarefa.class);
	}

	public IDAOWorkflow createDAOWorkflow() throws
DAOConcreteFactoryException{
		oLogger.info("CRIOU a DAOHibernateWorkflow");
		return
(IDAOWorkflow)getDAOInstance(DAOHibernateWorkflow.class);
	}

	public IDAOFuncionario createDAOFuncionario() throws
DAOConcreteFactoryException{
		oLogger.info("CRIOU a DAOHibernateFuncionario");
		return
(IDAOFuncionario)getDAOInstance(DAOHibernateFuncionario.class);
	}

	public IDAOPergunta createDAOPergunta() throws
DAOConcreteFactoryException{
		oLogger.info("CRIOU a DAOHibernatePergunta");
		return
(IDAOPergunta)getDAOInstance(DAOHibernatePergunta.class);
	}

	public IDAOPesquisaQualidade createDAOPesquisaQualidade() throws
DAOConcreteFactoryException{
		oLogger.info("CRIOU a DAOHibernatePesquisaQualidade");
		return
(IDAOPesquisaQualidade)getDAOInstance(DAOHibernatePesquisaQualidade.class);
	}

	public IDAOResposta createDAOResposta() throws
DAOConcreteFactoryException{
		oLogger.info("CRIOU a DAOHibernateResposta");
		return
(IDAOResposta)getDAOInstance(DAOHibernateResposta.class);
	}

	public IDAOOcorrencia createDAOOcorrencia() throws
DAOConcreteFactoryException{
		oLogger.info("CRIOU a DAOHibernateOcorrencia");
		return
(IDAOOcorrencia)getDAOInstance(DAOHibernateOcorrencia.class);
	}

	public IDAOSolucaoOcorrencia createDAOSolucaoOcorrencia() throws
DAOConcreteFactoryException{
		oLogger.info("CRIOU a DAOHibernateSolucaoOcorrencia");
		return
(IDAOSolucaoOcorrencia)getDAOInstance(DAOHibernateSolucaoOcorrencia.class);
	}
```

Quadro 10: Implementação da fábrica de *DAOs* do *Hibernate (Hibernate DAO Factory)* (Continuação)

```
private DAOHibernateGenerica getDAOInstance(Class cClass) throws
DAOConcreteFactoryException {
            try{
                  DAOHibernateGenerica oDAOHibernateGenerica =
(DAOHibernateGenerica)cClass.newInstance();
                  oDAOHibernateGenerica.setSession(getCurrentSession());
                  return (oDAOHibernateGenerica);
            }catch (Exception ex){
                  throw new DAOConcreteFactoryException("Impossível
instanciar o Data Access Object " + cClass);
            }
      }

      protected Session getCurrentSession() {
            return HibernateUtil.getSession();
      }
}
```

Quadro 11: Implementação da fábrica de *DAOs* do *Hibernate (Hibernate DAO Factory)* (Continuação)

Da mesma forma que foi criada a fábrica de *DAOs* para o Hibernate caso seja necessária a substituição do meio de armazenamento, ou a utilização de um meio de armazenamento paralelo, poderiam ser criadas fabricas para todos os meios de armazenamento necessários. Isto implicaria na criação de *DAOs* específicas para todos os meios de armazenamento, porém se estas implementarem a mesma interface *DAO*, estarão funcionando de acordo com o que a aplicação espera, e não serão necessárias alterações nas demais camadas da aplicação.

Embora a utilização das fábricas *DAO* já facilite algumas situações, sua utilização de forma pura ainda não traz o resultado esperado de desacoplamento do meio de armazenamento, pois precisaríamos nos referir claramente a fábrica que desejássemos utilizar. Para agrupar as fabricas em um componente único e que ofereça a abstração final ao meio de armazenamento, foi aplicado o *design pattern Abstract Factory*.

Este padrão introduz uma forma de encapsular um grupo de fábricas em uma determinada categoria, sendo estas programadas como implementações concretas da fábrica abstrata. Este padrão de desenho está a um nível de abstração bem elevado e pode ser usado quando é necessário retornar uma das muitas classes de objetos relacionadas, cada qual podendo retornar diferentes

objetos. De forma resumida, o padrão *Abstract Factory* é uma fábrica de objetos que retorna uma das várias fábricas concretas. Isto permite que os componentes da aplicação usem a fabrica abstrata para servir o tipo de fabrica concreta necessária.

A fábrica abstrata é uma classe *Java* abstrata. Uma classe abstrata é muito semelhante a uma interface e é desenvolvida para representar entidades e conceitos abstratos. A classe abstrata é sempre uma classe pai que não possui instâncias. A diferença entre uma interface e uma classe abstrata reside no fato de que a última além de definir um modelo para uma funcionalidade pode fornecer uma implementação incompleta – a parte genérica de uma funcionalidade – que é compartilhada por um grupo de classes derivadas. Cada uma das classes derivadas completa a funcionalidade da classe abstrata adicionando um comportamento específico. Uma classe abstrata normalmente possui métodos abstratos. Esses métodos são implementados nas suas classes derivadas concretas com o objetivo de definir o comportamento específico. O método abstrato define apenas a assinatura do método e, portanto, não contém código.

Foi criada na camada de persistência uma fábrica abstrata de *DAOs (DAO Abstract Factory)*, cuja função é padronizar e servir qualquer uma das fábricas concretas de *DAOs* para os clientes do serviço de persistência.

```
//Imports suprimidos

public abstract class DAOAbstractFactory {

	// Campos estáticos que fazem referência as classes que implementam
fábricas de façades
	public static final Class HIBERNATE =
br.gov.pr.ocorrencias.dao.DAOHibernateFactory.class;

	// Método estático para retornar a instância de uma fábrica de DAOs
	public static DAOAbstractFactory getInstance(Class sClass) throws
DAOAbstractFactoryException {
		try {
			return(DAOAbstractFactory)sClass.newInstance();
		} catch (Exception ex){
			throw new DAOAbstractFactoryException("Impossível criar
a fabrica de Data Access Objects " + sClass);
		}
	}
```

Quadro 12: A implementação da fábrica abstrata de *DAOs (DAO's Abstract Factory)*.

```
// Assinatura dos métodos de criação de cada uma das DAOs de uma fábrica
      public abstract IDAOFormaAtendimento createDAOFormaAtendimento()
throws DAOConcreteFactoryException;
      public abstract IDAOGrauUrgencia createDAOGrauUrgencia() throws
DAOConcreteFactoryException;
      public abstract IDAOOrgao createDAOOrgao() throws
DAOConcreteFactoryException;
      public abstract IDAOAreaAbrangencia createDAOAreaAbrangencia()
throws DAOConcreteFactoryException;
      public abstract IDAOPessoa createDAOPessoa() throws
DAOConcreteFactoryException;
      public abstract IDAOContato createDAOContato() throws
DAOConcreteFactoryException;
      public abstract IDAOSolicitante createDAOSolicitante() throws
DAOConcreteFactoryException;
      public abstract IDAOTipoSolicitante createDAOTipoSolicitante()
throws DAOConcreteFactoryException;
      public abstract IDAOItemHistorico createDAOItemHistorico() throws
DAOConcreteFactoryException;
      public abstract IDAOTipoOcorrencia createDAOTipoOcorrencia() throws
DAOConcreteFactoryException;
      public abstract IDAOSolucaoOcorrencia createDAOSolucaoOcorrencia()
throws DAOConcreteFactoryException;
      public abstract IDAOFuncionario createDAOFuncionario() throws
DAOConcreteFactoryException;
      public abstract IDAOPesquisaQualidade createDAOPesquisaQualidade()
throws DAOConcreteFactoryException;
      public abstract IDAOOcorrencia createDAOOcorrencia() throws
DAOConcreteFactoryException;
      public abstract IDAOPergunta createDAOPergunta() throws
DAOConcreteFactoryException;
      public abstract IDAOResposta createDAOResposta() throws
DAOConcreteFactoryException;
      public abstract IDAOUsuario createDAOUsuario() throws
DAOConcreteFactoryException;
      public abstract IDAOPerfil createDAOPerfil() throws
DAOConcreteFactoryException;
      public abstract IDAOWorkflow createDAOWorkflow() throws
DAOConcreteFactoryException;
      public abstract IDAOTarefa createDAOTarefa() throws
DAOConcreteFactoryException;
}
```

Quadro 13: A implementação da fábrica abstrata de *DAOs (DAO's Abstract Factory)* (Continuação).

Esta fábrica é composta de um método estático para retornar a instância de uma fábrica de *DAOs*, por campos estáticos que fazem referência as classes que implementam fábricas de *DAOs* e por uma série de métodos abstratos que definem a assinatura dos métodos de criação de cada uma das *DAOs* de uma fábrica. Ou seja, estes métodos abstratos, atuam de forma análoga as interfaces

das *DAOs*, definido quais métodos as fábricas de *DAOs* são obrigadas a implementar.

Através desta estrutura, para um fábrica de *DAOs* passe a atuar como uma fábrica concreta e fazer parte da camada de persistência da aplicação basta que ela seja registrada em um campo estático na fábrica abstrata e derive esta última, implementando os métodos abstratos que esta define para retornar as *DAOs* de seu respectivo meio de armazenamento.

Portanto, com a aplicação do *design pattern Factory* gerenciado pelo *design pattern Abstract Factory* tornou-se finalmente possível abstrair a camada de persistência do meio de armazenamento utilizado, estando esta camada estruturada e preparada para quaisquer modificações futuras sem quaisquer comprometimentos de seus clientes.

3.2.3.4 Criando uma Entidade Única para Acesso aos Serviços de Persistência

Como resultado da aplicação do *design pattern Abstract Factory* também foi possível à centralização dos serviços de persistência em uma entidade única: a fábrica abstrata de *DAOs*. Todo e qualquer objeto da camada de modelo que deseje utilizar o serviço de persistência deverá utilizar a fábrica abstrata como um portão de entrada da camada de persistência. A utilização do serviço de persistência é executada conforme o exemplo do quadro abaixo.

```
FormaAtendimento oFormaAtendimento
       = DAOAbstractFactory.getInstance(DAOAbstractFactory.HIBERNATE).
            createDAOFormaAtendimento().get(codigo);
```

Quadro 14: Exemplo de utilização do serviço de persistência.

Um objeto é instanciado através do acesso a fábrica abstrata, que irá retornar uma instância de um fábrica concreta – no caso uma fábrica concreta de

DAOs para utilização do *Hibernate* –, que irá retornar uma interface para uma *DAO* e esta última irá executar o método solicitado pelo cliente.

3.2.3.5 A Generalização das Operações Básicas de Persistência *(CRUD)*

O resultado da aplicação do *design pattern DAO* foi à criação de objetos de acesso a dados para todos os objetos de negócio da aplicação. Nestas *DAOs*, foi possível analisar que os métodos que implementam as operações básicas de persistência – *CRUD (Create, Read, Update, Delete)* – possuem lógica idêntica, alterando-se apenas o tipo de objeto manipulado. Esta situação favorece que trechos de código sejam copiados e colados de uma *DAO* para outra, alterando-se apenas o tipo de objeto a ser tratado. A prática de cópia de código dificulta a manutenção e desconsidera por total o princípio da reutilização de código.

Para maximizar a reutilização de código e reduzir o esforço de manutenção das *DAOs* foi aplicado o *design pattern Generic DAO*. Este padrão, como o seu nome sugere, consiste da aplicação do *design pattern DAO* utilizando o recurso *generics*, disponível na linguagem *Java* a partir da plataforma versão 5.0. Este é um recurso de programação poderoso que possibilita que os algoritmos sejam escritos de forma a manipular tipos genéricos de objetos. Estes algoritmos são escritos em uma sintaxe extensível, ou seja, que pode ser herdada, em que as classes especializadas (classes filhas) são capazes de adaptarem-se a lógica estipulada na classe genérica (classe pai) com a condição de especificar qual deverão ser os tipos de objeto a serem manipulados. Ou seja, a classe genérica é responsável por criar um modelo de implementação e a classe especializada por apontar que tipos de objetos serão manipulados neste modelo. No ato da herança, a classe especializada irá incorporar os métodos da classe genérica substituindo os tipos de objeto genéricos pelos tipos que ela irá manipular.

Com isso, ao invés de replicarmos o mesmo código de implementação das operações de *CRUD* para cada *DAO* de nossa camada de persistência, apenas definimos uma classe genérica – com o algoritmo genérico e os tipos genéricos – e lançamos mão do uso da herança para as demais *DAOs*, que em tempo de

compilação, são obrigadas a fornecer os tipos de objetos que deverão ser utilizadas pelo algoritmo genérico.

Isto permite que as operações triviais e repetitivas de *CRUD* sejam programadas em uma classe genérica (independente de tipos) que será herdada por todas as demais *DAOs* especializadas. Estas ao especializarem-se englobam os métodos da primeira estipulando os tipos reais a serem tratados, podendo também implementar novos métodos se necessário.

3.3 A Camada de Modelo

A camada de modelo é o núcleo de funcionamento da aplicação, sendo responsável por reunir os objetos responsáveis em armazenar os dados e o comportamento do negócio. Ela é essencialmente composta pelos objetos de negócio, que interagem para a concretização dos casos de uso.

Os objetos de negócio foram criados com base no diagrama de classes que é fruto da modelagem orientada a objetos.

Esta camada implementa o *design pattern façade*, através do qual são definidas “fachadas” para acesso aos seus recursos, garantindo mais segurança e modularidade.

3.3.1 A modelagem

A modelagem da camada de modelo foi realizada abrangendo os casos de uso e o diagrama de classes abaixo ilustrados:

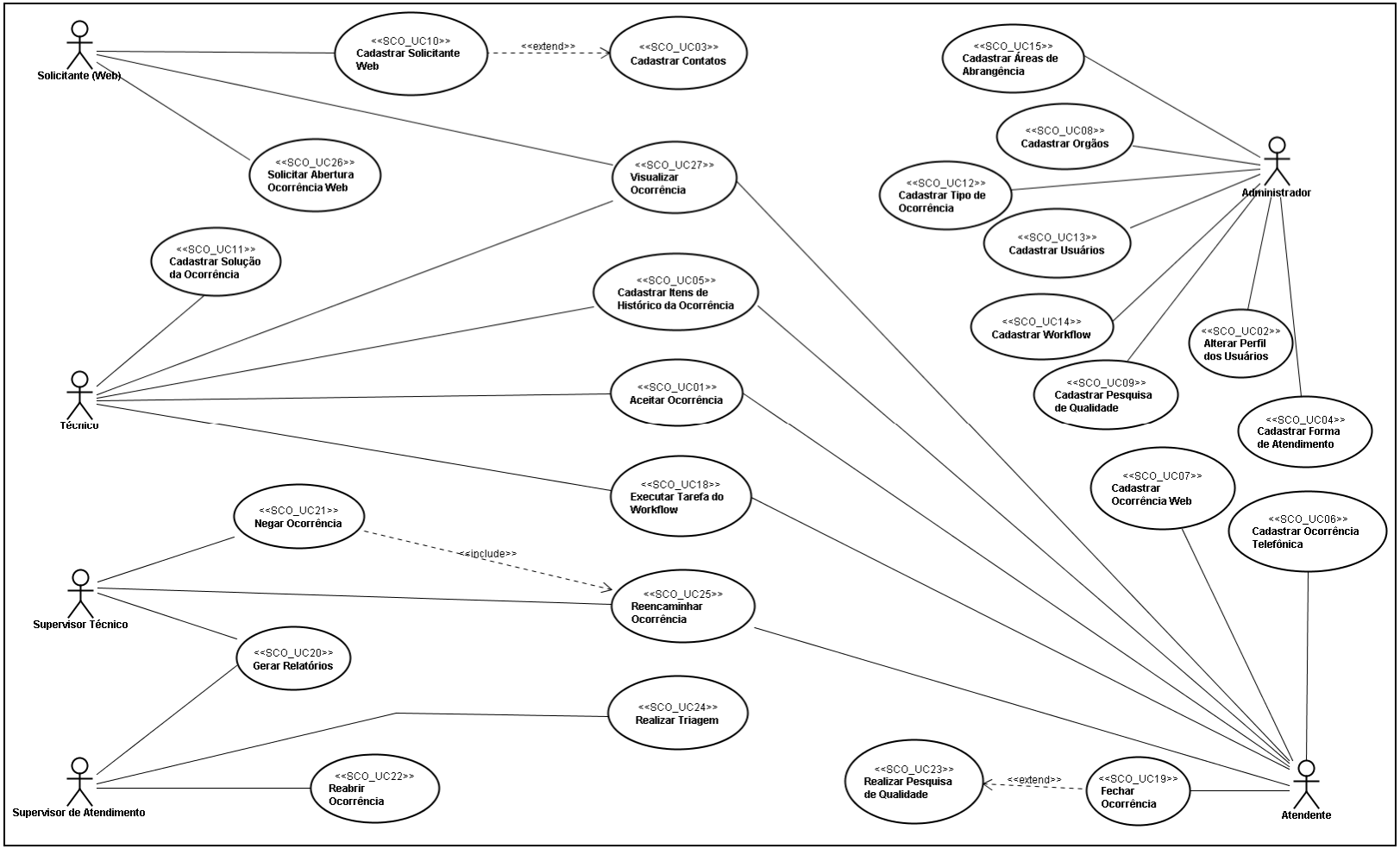


Figura 11: Diagrama de Casos de Uso

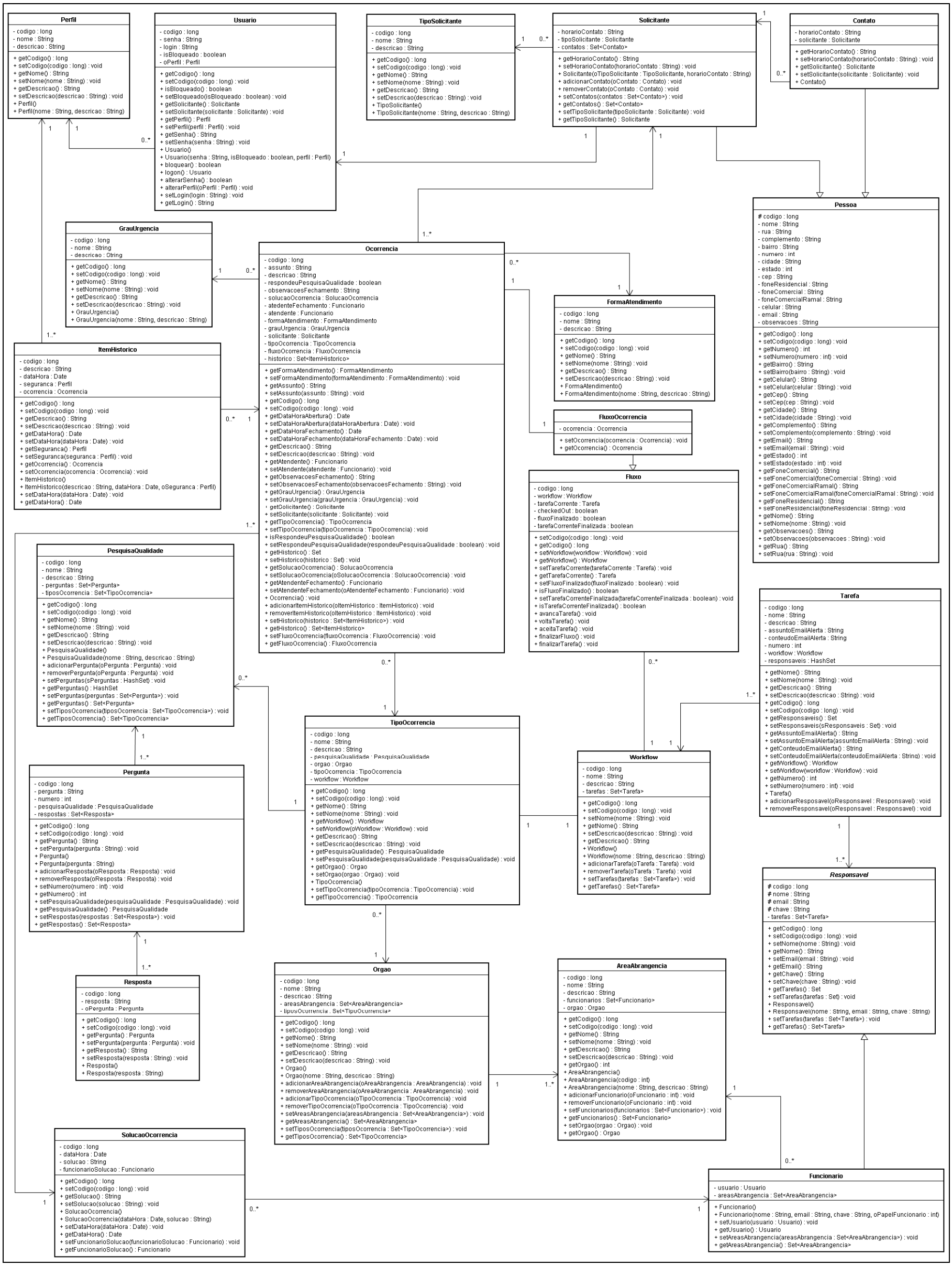


Figura 12: Diagrama de classes da camada de modelo.

3.3.2 Escondendo a complexidade da Camada de Modelo: Façades

Utilizando como base o diagrama de classes, foi implementada a camada de modelo. Porém, simplesmente deixá-la exposta não é a melhor alternativa, principalmente se quisermos:

- Reduzir as dependências em relação às características internas da camada, trazendo flexibilidade no desenvolvimento da aplicação.
- Isolar as alterações na camada de modelo dos demais componentes da aplicação.
- Que a camada de modelo seja mais fácil de entender e usar.
- Que o código que utiliza esta camada seja mais fácil de compreender.

Para atender os tópicos acima foi utilizado um padrão de desenho que auxilia na implementação destes requisitos: o *design pattern Façade*.

A palavra *façade* é proveniente do francês e geralmente é utilizada quando nos referimos a fachada de uma construção. Quando olhamos a fachada de um prédio nós não visualizamos como este foi construído, suas colunas, cabeamento ou outras complexidades. De forma análoga deve estar estruturada a camada de modelo da aplicação, pois seus clientes não devem saber detalhes de como ela foi implementada.

As *façades* não possuem nenhum padrão de implementação, devem apenas atender ao princípio de definir uma *interface* de alto nível para facilitar o uso de um subsistema, no caso, a camada de modelo. Desta forma, a *façade* é uma classe *Java* comum, geralmente criada com base na solução de um caso de uso, ou conjunto de casos de uso.

Na camada de modelo do SCO foram criadas diversas *façades* responsáveis pelo gerencia da segurança da aplicação, dos solicitantes, dos tipos de ocorrência, dos órgãos, das pesquisas de qualidade, dos cadastros auxiliares e das ocorrências. Cada uma destas abstrai os relacionamentos e dependências entre os objetos de negócio envolvidos na realização de um ou mais casos de uso.

Isto resulta em menos invocações a métodos, maior simplicidade na *interface* de serviços para o cliente e contribui para o desacoplamento entre a

camada de modelo e seus clientes, uma vez que a implementação da camada de modelo fica transparente.

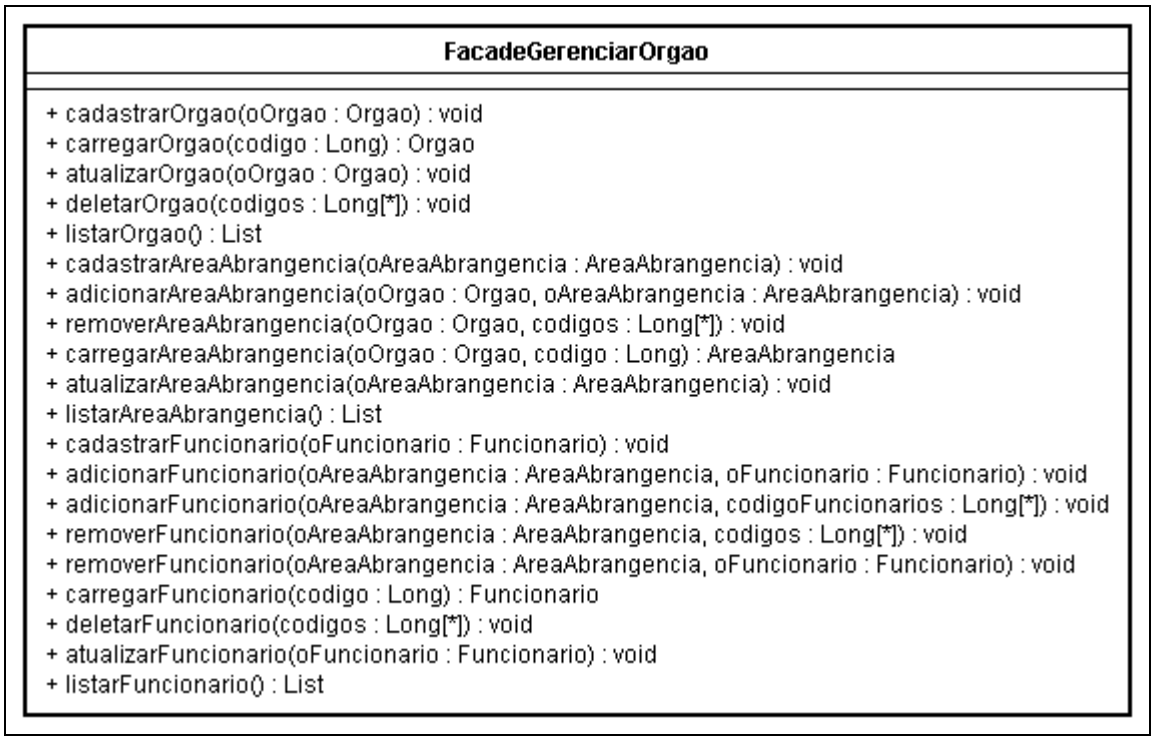


Figura 13: Exemplo de uma classe *Façade* do SCO.

3.3.3 Criando uma Entidade Única para Acesso a Camada de Modelo

Para criação de uma entidade única para acesso aos serviços da camada de modelo, assim como na camada de persistência, foram aplicados os *design patterns Factory* e *Abstract Factory*.

Foram criadas interfaces para cada *façade* da camada de modelo e foi desenvolvida uma fábrica *(SCO Concrete Factory)* responsável por gerir qual *façade* concreta deverá ser fornecida.

Esta fábrica foi construída de forma a oferecer métodos para a criação das *façades* da camada de modelo. Ela é responsável por implementar métodos que retornem a interface para uma *façade* concreta. Esta *façade* concreta pode ser escolhida pelo método com base em parâmetros que ele tenha recebido, ou o método pode não possuir nenhuma lógica e simplesmente retornar uma instância da *façade* concreta. O que é importante perceber é que foi definido um local central responsável pela forma como as *façades* são criadas. Neste local podem ser implementadas os mais variados tipos de

funções, como o rastreamento do uso de *façades* através de *logs,* implementação de segurança e tudo mais que esteja ao alcance da imaginação do programador. Outro aspecto importante é que na criação de uma *façade*, não é necessário à utilização do comando *new*, pois a fábrica já retorna uma instancia do objeto concreto como uma interface. Caso fosse necessária uma manutenção no nome das *façades* ou até mesmo a substituição de uma *façade* por outra, a alteração na fábrica faria valer para todos os clientes da camada de modelo, diminuindo consideravelmente o esforço de manutenção.

Depois de criada a fábrica de *façades* para podermos acoplar diferentes famílias de *façades* a camada de modelo e não precisarmos nos referir claramente a fábrica que desejamos, foi aplicado o *design pattern Abstract Factory*.

Como resultado, foi criada uma fábrica abstrata de *façades (SCO Abstract Factory)*, cuja função é padronizar e servir qualquer uma das fábricas concretas de *façades* para os clientes camada de modelo.

Esta fábrica é composta de um método estático para retornar a instância de uma fábrica de *façades*, por campos estáticos que fazem referência as classes que implementam fábricas de *façades* e por uma série de métodos abstratos que definem a assinatura dos métodos de criação de cada uma das *façades* de uma fábrica.

Através desta estrutura, para um fábrica de *façades* passe a atuar como uma fábrica concreta e fazer parte da camada de modelo da aplicação basta que ela seja registrada em um campo estático na fábrica abstrata e derive esta última, implementando os métodos abstratos que esta define para retornar as *façades.*

Como resultado da aplicação do *design pattern Abstract Factory* foi possível à centralização do acesso a camada de modelo em uma entidade única: a fábrica abstrata de *façades*. Todo e qualquer cliente da camada de modelo que deseje utilizá-la deverá utilizar a fábrica abstrata como um portão de entrada.

3.4 A Camada Controladora

A camada controladora é responsável por interpretar as requisições provenientes da camada de apresentação, comandar a execução das regras

de negócio na camada de modelo, obter desta os dados que devem ser disponibilizados pela camada de apresentação e comandar a apresentação do conteúdo na *interface* gráfica. Para cumprir estas determinações o desenvolvimento desta camada foi realizado utilizando *Struts Framework*.

3.4.1 Implementação da Camada Controladora

O desenvolvimento da camada controladora foi realizada acima do *Struts*, um *framework* que fornece uma implementação do *Sun´s Model 2* para a criação de aplicações *web*.

O fluxo normal de uma aplicação *Struts* e os componentes envolvidos em cada momento de seu funcionamento são demonstrados na figura abaixo:

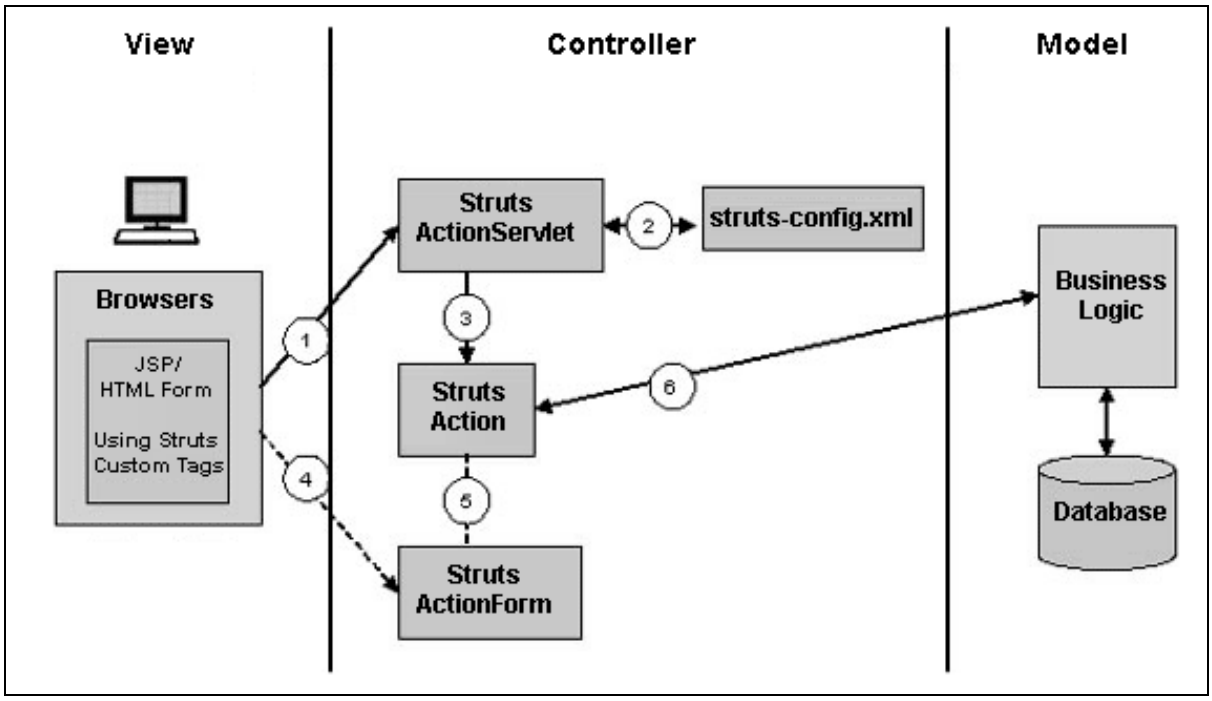


Figura 14: Funcionamento do *Struts Framework*.

1. A ativação do fluxo de funcionamento do *framework* é realizada através de uma solicitação *HTTP* a uma *URL* que esteja sob o domínio do *ActionServlet*, normalmente estas *URLs* tem como sufixo os caracteres ".do".
2. Cada *URL* que será tratada pelo *Struts* é mapeada a uma *Action* através do arquivo de configuração do *framework*, o *struts-config.xml*,

3. Quando é feita uma requisição o *ActionServlet* seleciona a *Action* correspondente a *URL* requisitada, e esta pode executar as operações programadas em sua definição.
4. Uma requisição *HTTP* pode ser realizada, também, através da submissão de um formulário *HTML* a uma *URL*. Quando isto acontece, o *framework* automaticamente encapsula os valores de cada campo do formulário em um *JavaBean*, chamado *FormBean*,que é repassado a Action responsável por seu processamento.
5. A *Action* pode, então, acessar o *FormBean* e manipulá-lo da forma que foi programada.
6. Por fim, após manipular os dados do FormBean a Action é capaz de interagir com a camada de modelo, através da qual a base de dados pode ser atualizada ou consultada para exibir algum dado em tela.

Em geral, o *ActionServlet* não fornece a resposta de uma requisição em si, ele é responsável por delegar a solicitação a outros recursos, como uma página *JSP* ou uma *Action*.

Os *links* dentro do *Struts* são feitos através de uma classe chamada *ActionForward*, que armazena o caminho para uma página sob um nome lógico [HUSTED et al, 2004], ocultando do usuário o endereços reais a serem utilizados pelo *framework*. Desta forma, ao completar uma lógica de negócio, a *Action* selecionará e retornará um *ActionForward* para o *ActionServlet*. Este usará o caminho armazenado no objeto *ActionForward* para chamar a página especificada e completar a resposta da requisição, seja ela uma operação de sucesso ou erro.

O conjunto destes detalhes logísticos da aplicação são definidos em um objeto chamado *ActionMapping*. Cada ActionMapping está relacionado a uma *URL* específica. Quando esta *URL* é requisitada, o *ActionServlet* irá recuperar o objeto *ActionMapping* correspondente, e através dele saberá quais *Actions* , *ActionForms* e *ActionForwards* deverão ser utilizados para tratar a requisição.

A partir deste panorama, para o desenvolvimento da camada controladora do SCO foi necessário criar os diversos *ActionMappings* para todas as operações que a aplicação deveria suportar. Estes *ActionMappings* foram definidos no arquivo *XML* descritor do *framework*: o *struts-config.xml*.

Acompanhados dos *ActionMappings* foram criadas as *Actions,* que serão os componentes responsáveis por interagir com as *façades* da camada de

modelo para prover as funcionalidades da aplicação. Da mesma forma, foram criados os *FormBeans* responsáveis por encapsular os dados provenientes do formulários *HTML*.

Uma vez estas tarefas realizadas o restante é executado pelo *Struts Framework*.

3.5 A Camada de Apresentação

A camada de apresentação é responsável pela exibição dos dados em tela. Esta camada foi desenvolvida utilizando as metodologias *tableless* e *W3C Web Standards* juntamente com as tecnologias *HTML, JSP, Tag Libraries, CSS e Javascript*.

Atualmente o *HTML* é a linguagem mais utilizada no desenvolvimento de interfaces com o usuário em aplicações baseadas na *web*. O seu uso traz diversas vantagens:

1. A aplicação é instalada no servidor. O usuário usa o navegador instalado em seu computador que funciona como um cliente universal.
2. Desacopla o usuário da tecnologia usada no servidor.
3. Os firewalls são configurados para liberar o tráfego de rede na porta utilizada pelo servidor de páginas *HTML*, diminuindo o esforço de configuração.
4. Impõem restrições de configurações de hardware mais modestas às máquinas dos usuários já que a maior parte do processamento ocorre no servidor.

Contudo, o desenvolvimento de aplicações baseadas na internet com *HTML* e o protocolo *HTTP* impõe uma série de desafios:

1. Uma interface *HTML* é limitada pelo modelo de requisição e resposta, diferentemente de outras tecnologias de construção de interface como o *Swing*, usada comumente em aplicações *Java desktop*, onde um componente de interface pode ser atualizado automaticamente quando ocorre alguma mudança no modelo.
2. Interfaces *web* costumam apresentar marcações complexas como tabelas aninhadas e extensos trechos de código *JavaScript* sendo que apenas uma pequena parte desta marcação se destina a conteúdo gerado dinamicamente. O código de marcação que constrói o layout da

interface deve ser separado do código que efetua as operações de negócio de forma a possibilitar a separação em papéis: desenvolvedor e *web designer*.

3. Interfaces *HTML* tornam a validação da entrada do usuário mais importante, pois a aplicação tem controle limitado sobre o navegador o qual o usuário insere os dados;
4. *HTML* oferece um conjunto limitado e não expansível de componentes de interface.
5. Garantir que uma aplicação tenha o mesmo visual em todos os navegadores é custoso, devido as variações na aderência aos padrões *HTML, JavaScript* e *CSS*;

Para garantir a utilização de dados dinâmicos e garantir a separação de atividades entre o desenvolvedor e o *web desginer* foram utilizados o *JSP* e as *Tag Libraries*. Estas duas tecnologias possibilitam a construção de páginas *JSP* com a clássica sintaxe *HTML* para a exibição de elementos estáticos e quando é necessária a exibição de dados dinâmicos são utilizadas as *tag libraries*, que podem ser misturadas as *tags HTML* como se fizessem parte de uma mesma sintaxe.

A utilização das metodologias *tableless* e *W3C Web Standards* e, portanto obrigatoriamente a utilização do *CSS*, auxiliou na garantia de que a aplicação tenha o mesmo visual em todos os navegadores de mercado.

A linguagem *JavaScript* auxiliou na validação da entrada de dados do usuário no lado cliente.

A divisão da aplicação em camadas, com uma delas destinada à apresentação, contribui para que esta possa ser alterada sem que a de negócios sofra alterações, pois uma camada só depende de outra imediatamente inferior. Para que isto ocorra é preciso também que a camada de apresentação seja limpa, isto é, o fluxo de controle e a chamada dos métodos de negócios devem ser separados da visão.

Porém, com a atual arquitetura do SCO podemos perceber claramente que a camada de controle, juntamente com as *façades* solucionam mais este impasse.

3.6 Resultado do Processo: O produto

O Sistema de Controle de Ocorrências foi utilizado como base para a aplicação do objetivo principal deste projeto. Sem um desafio claro, de desenvolvimento de software, seria impossível simular de forma real a aplicação dos diversos *design patterns, lightweight frameworks, W3C Web Standards* e demais ferramentas e metodologias.

O produto final, embora inacabado, atende em partes as necessidades da CELEPAR em registrar e atender, de forma rápida e eficiente, as solicitações de serviços geradas a partir de seus clientes internos ou externos.

A solução desenvolvida possui um código-fonte robusto, reutilizavél, inteiramente orientado a objetos, seguro, de manutenção simplificada, eficiente e eficaz, tornando trivial a finalização da aplicação e a implementação de módulos futuros que venham a surgir.

4 CONCLUSÃO

O desenvolvimento aplicações normalmente é componente crítico da missão de qualquer empresa. As equipes de desenvolvimento precisam construir aplicações em curto espaço de tempo, porém tem que construí-las corretamente, de forma a facilitar sua manutenção e maximizar o reuso de código.

A utilização de *lightweight frameworks* como o *Struts* e o *Hibernate* auxiliam as equipes de desenvolvimento nesta missão.

O *Struts* possibilita o desenvolvimento de aplicações *web*, utilizando-se do *Sun's Model 2*. Este modelo prevê uma definição bem clara das camadas em uma aplicação, permitindo a distribuição do trabalho, de forma que o *web designer* possa concentrar-se unicamente no desenvolvimento das páginas *JSPs*, enquanto o restante da aplicação está na vanguarda dos desenvolvedores, analistas de sistemas e arquitetos de *software*. Ele auxilia na redução do acoplamento entre as páginas *JSP* permitindo que a o fluxo da aplicação seja altamente flexível.

O *Hibernate* facilita o desenvolvimento de aplicações que acessam bancos de dados relacionais, permitindo aos desenvolvedores preocuparem-se com a consistência de sua camada de modelo (regras de negócio) ao invés de gastarem tempo no desenvolvimento dos mecanismos de persistência de dados.

Afora os *lightweight frameworks,* a utilização de *design patterns* encaixa-se perfeitamente no processo de desenvolvimento de *software*. Eles incentivam a formação de um vocabulário comum para uma melhor comunicação entre os desenvolvedores, a maior exploração das alternativas de projeto, redução da complexidade de entendimento da aplicação através da definição de abstrações, constituição de uma base de experiências reutilizáveis para a construção de software, atuação como peças na construção de projetos de software mais complexos podendo ser considerados como micro-arquiteturas

que contribuem para arquitetura geral do sistema, reduzem o tempo de aprendizado de uma determinada biblioteca de classes e quanto mais cedo são usados, menor será o re-trabalho em etapas mais avançadas do projeto.

Concluímos que a união entre *design patterns* e *lightweight frameworks* permitem sem dúvida alguma o desenvolvimento de aplicações de fácil manutenção, portáveis, de componentes rastreáveis, de alta reusabilidade de código e altamente modularizadas.

5 REFERÊNCIAS BIBLIOGRÁFICAS

GAMMA, Enrich; HELM, Richar; JHONSON, Ralph; VLISSIDES, John, *Design patterns*, Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1995.

HUSTED, Ted N.; DUMOULIN, Cedric; FRANCISCUS, George; WINTERFELDT, David, *Struts in Action, Building web applications with the leading Java framework*. Manning Publications Co., 2002.

RICHARDSON, Chris, *POJOs in Action, Developing Enterprise Applications With Lightweight Frameworks*. Manning Publications Co., 2006.

BAYERN, Shawn, *JSTL in Action*. Manning Publications Co., 2003.

BAUER, Christian; KING, Gavin, *Hibernate in Action*. Manning Publications Co., 2005.

QUADROS, Moacir, *Gerência de Projetos, Técnicas e Ferramentas*. Florianópolis:Visual Books, 2002.

PRESSMAN, Roger S, *Engenharia de software*. São Paulo: Pearson Makron Books, 1995.

ECKEL, Bruce, *Thinking in Java*. Pearson Education, Inc, 2003.

RUMBAUGH, James, *UML Guia do Usuário*. Campus.
INC, Sun Microsystems.

FOUNDATION, The Apache *Software*, *Struts*. http://struts.apache.org/struts-doc-1.2.7/userGuide/index.html. Acesso em 05/09/2005.

CORPORATION,JasperSoft, *Jasper Reports Documentation*. http://jasperreports.sourceforge.net/documentation.html. Acesso em 15/05/2006.

GARNIER, Jean-Michel, *Struts 1.1 Controller UML diagrams*. França. http://rollerjm.free.fr/pro/Struts11.html#3. Acesso em 25/03/2006.

INC, Sun Microsystems, *jGuru: Fundamentals of the JavaMail API*. Sun Developer Network.
http://java.sun.com/developer/onlineTraining/JavaMail/contents.html. Acesso em 18/03/2006.

GUJ. *Java e Desenvolvimento Web*, http://www.guj.com.br/forums/list.java. Acesso em 02/03/2006.

FREITAS, Katiúcia, *Criar uma aplicação utilizando Eclipse, Struts e o FrameWork DBFW4J.* Imasters – Programação Java. http://www.imasters.com.br/artigo/3336. Acesso em 24/04/2006.

MICHELAZZO, Paulino, *A liberdade dos bancos de dados.* Imasters – *Software* Livre. http://www.imasters.com.br/artigo/3990/livre/a_liberdade_dos_bancos_de_dados. Acesso em 24/04/2006.

ALECRIM, Emerson, *Banco de dados PostgreSQL e MySQL***.** Info Wester. http://www.infowester.com/postgremysql.php. Acesso em 24/04/2006.

JEVEAUX, Matheus, *Incremente suas aplicações Web com Servlets e Java Server Pages***.** Portal Java. http://www.portaljava.com.br/home/modules.php?name=News&file=article&sid=41. Acesso em 25/04/2006.

DEBONI, José Eduardo Zindel, *Breve Introdução aos Diagramas da UML.* Voxxel – Conceito em Tecnologia da Informação desde 1988. http://www.voxxel.com.br/pages/introdiauml.html#estados. Acesso em 230/03/2006.

ALVES, Maria Bernardete Martins; ARRUDA, Susana Margareth, *Elaboração Referenciais*. UFSC – New Page 8. Brasil. http://www.bu.ufsc.br/framerefer.html. Acesso em 26/08/2006.

ZANETTI, Eni Maria de Souza Pinto, *Como fazer referências: bibliográficas, eletrônicas e demais formas de documentos*. FUNCAB. Brasil. http://www.funcab.br/Paginas/referencias.htm. Acesso em 26/08/2006.

CONTE, S.D, *Software engineering metrics and models*. Califórnia: Benjamin/Cummings Publishing, 1985.

CAVANNES, Chuck, *Programming Jakarta Struts*. Greenwich: O ' Reilly Publications, 2003.

DUDNEY, Bill, LEHR, Jonathan, *Jakarta Pitfalls - Time-saving Solutions for Struts, ANT, Junit and Cactus*. Indianapolis: Willey Publishing. 2003.

APÊNDICES

APÊNDICE I – PLANO GERAL DE PROJETO

Sistema de Controle de Ocorrências

Plano de Projeto

1. Introdução

1.1 Escopo e Propósito do Documento

O presente documento tem o objetivo de apresentar um planejamento para um Sistema de Controle de Ocorrências (SCO) que faz o acompanhamento das ocorrências técnicas ou não técnicas, internas ou externas de uma empresa.

O sistema será desenvolvido em parceria com a empresa CELEPAR e funcionará em ambiente w*eb*. A linguagem utilizada será o *Java* e banco de dados *PostgreSql*.

Este documento é constituído pelo detalhamento dos objetivos do projeto, uma análise de risco, cronogramas, estimativas, organização da equipe bem como as principais funções que serão desempenhadas pela ferramenta.

1.2 Escopo do software

O sistema cadastra as ocorrências através do contato de um usuário via telefone ou internet.

No contato via telefone o SCO coletará automaticamente o número de telefone do usuário através de um componente para a leitura de um aparelho identificador de chamada (bina), verificando a possível existência do usuário no banco de dados, e dando procedimento ao cadastro da ocorrência. No contato via internet o sistema irá cadastrar o usuário, caso ele não exista, e a ocorrência.

Após estes cadastros o SCO redirecionará a ocorrência, via e-mail, para o gestor da área de abrangência responsável pela solução desta. O sistema permitirá ao gestor apontar um ou mais técnicos para a análise e resolução da ocorrência.

Uma vez solucionada a ocorrência o SCO cadastrará a solução explorada pelos técnicos. Caso o atendimento tenha ocorrido via telefone o sistema fornecerá aos técnicos responsáveis às informações do solicitante para que este seja informado da solução e o chamado é então finalizado. Caso o atendimento tenha ocorrido via internet o solicitante será informado da solução via e-mail e o chamado será finalizado.

No caso do atendimento telefônico, estando à ocorrência finalizada, o SCO a fará voltar, via e-mail, para o atendente inicial ou para o atendente com menor fluxo de trabalho. Na seqüência o sistema disponibilizará as informações do solicitante para que o atendente entre em contato e cadastre a satisfação do cliente no atendimento.

No caso do atendimento via internet o SCO disponibilizará ao usuário uma área para a avaliação do atendimento.

O solicitante poderá acompanhar o andamento de sua ocorrência a qualquer momento do processo.

1.2 Objetivos

Objetivos do Projeto
Documentar as ocorrências internas/externas
Assegurar a melhor assistência técnica à ocorrência.
Manter o solicitante informado sobre a situação de sua ocorrência.
Acumular o conhecimento realizado no atendimento.
Melhoria na qualidade do atendimento
Melhoria do aproveitamento dos recursos humanos responsáveis pela resolução das ocorrências

1.2.1 Funções Principais

Módulo 1: Cadastro de ocorrências:

1. Controle de ocorrências;

Módulo 2: *Workflow* das ocorrências:

1. Controle de *workflow*;

Módulo 3: Fechamento das ocorrências:

1. Controle de avaliações de atendimento;

Módulo 4: Banco do conhecimento (Adicional)

1. Controle de documentos do banco de conhecimento;

Módulo 5: Administração

1. Controle de Usuários;

Módulo 6: Relatórios

1. Controle de relatórios;

2. Estimativas de Projeto

2.1 Dados históricos usados nas estimativas

Os dados históricos que servirão de base para realizar as estimativas foram extraídos da revista técnica *Software Development* de outubro de 2000. A revista apresenta a seguinte análise:

Linguagem de Programação	LOC por FP
C++	53
Cobol	107
Delphi 5	18
HTML 4	14
Visual Basic 6	24
SQL	13
Java	46

Estes dados serão utilizados para conversão de pontos-por-função (F.P. - *Functions Points*) em linhas de código (LOC – *Lines of Code*) nas análises de estimativa.

2.2 Técnicas de estimativas

Foram utilizadas as técnicas de pontos-por-função (F.P. - *Functions Points*) e LOC (*Lines of code*).

A métrica de pontos-por-função foi primariamente proposta por Albrecht [ALB79] e é baseada no fato de que o tamanho dos sistemas não depende das tecnologias empregadas, mas sim da especificação ou “funcionabilidade”.

Já a técnica de LOC – linhas de código – consiste em estimar o tamanho do software em linhas de código a serem programadas. Embora seja uma das métricas que mais se aproximam do real esforço a ser realizado para a confecção de um projeto, é altamente dependente da linguagem de programação utilizada.

Desta forma, ambas as técnica serão empregadas para gerar as estimativas deste projeto.

2.3 Estimativas

Estimativas em pontos-por-função das principais funções do SCO:

Módulo 1: Cadastro de ocorrências

1. Controle de ocorrências;

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**5**	X	3	4	6	**6**	=	30
Número de saídas do usuário	**0**	X	4	6	7	**4**	=	0
Número de consultas do usuário	**5**	X	3	4	6	**3**	=	15
Número de arquivos	**5**	X	7	10	15	**15**	=	75
Número de interfaces externas	**1**	X	5	7	10	**10**	=	10
						Total	=	130

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação

$FP = Total \times [0,65 + 0,01 \times SOMA(F_i)]$ = 128,7 FP

Módulo 2: *Workflow* das ocorrências;

1. Controle de *workflow*

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**3**	X	3	4	6	**6**	=	18
Número de saídas do usuário	**0**	X	4	6	7	**4**	=	0
Número de consultas do usuário	**3**	X	3	4	6	**6**	=	18
Número de arquivos	**3**	X	7	10	15	**15**	=	45
Número de interfaces externas	**0**	X	5	7	10	**0**	=	0
						Total	=	81

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação		
$FP = Total \times [0,65 + 0,01 \times SOMA(F_i)]$	=	80,19 FP

Módulo 3: Fechamento das ocorrências

1. Controle de avaliações de atendimento;

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**3**	X	3	4	6	**6**	=	18
Número de saídas do usuário	**0**	X	4	6	7	**7**	=	0
Número de consultas do usuário	**3**	X	3	4	6	**6**	=	18
Número de arquivos	**3**	X	7	10	15	**15**	=	45
Número de interfaces externas	**0**	X	5	7	10		=	0
						Total	=	81

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação		
$FP = Total \times [0,65 + 0,01 \times SOMA(F_i)]$	=	80,19 FP

Módulo 4: Banco do conhecimento (Adicional)

1. Controle de documentos do banco de conhecimento;

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**1**	X	3	4	6	**6**	=	6
Número de saídas do usuário	**0**	X	4	6	7	**4**	=	0
Número de consultas do usuário	**4**	X	3	4	6	**3**	=	12
Número de arquivos	**1**	X	7	10	15	**15**	=	15
Número de interfaces externas	**1**	X	5	7	10	**10**	=	10
						Total	=	43

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação		
$FP = Total \times [0,65 + 0,01 \times SOMA(F_i)]$	=	42,57 FP

Módulo 5: Administração

1.Controle de usuários;

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**2**	X	3	4	6	**3**	=	6
Número de saídas do usuário	**0**	X	4	5	7	**4**	=	0
Número de consultas do usuário	**2**	X	3	4	6	**3**	=	6
Número de arquivos	**2**	X	7	10	15	**10**	=	20
Número de interfaces externas	**1**	X	5	7	10	**10**	=	10
						Total	=	42

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação		
FP = Total x [0,65 + 0,01 x SOMA(F_i)]	=	41,58 FP

Módulo 6: Relatórios

1.Controle de relatórios;

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**1**	X	3	4	6	**6**	=	6
Número de saídas do usuário	**8**	X	4	6	7	**7**	=	56
Número de consultas do usuário	**8**	X	3	4	6	**6**	=	48
Número de arquivos	**0**	X	7	10	15	**7**	=	0
Número de interfaces externas	**2**	X	5	7	10	**10**	=	20
						Total	=	130

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação		
$FP = Total \times [0,65 + 0,01 \times SOMA(F_i)]$	=	128,7 FP

Totalização (por módulos/geral)

Módulo 1 – Cadastro de ocorrências	128,70 FP
Módulo 2 – Workflow das ocorrências	80,19 FP
Módulo 3 – Fechamento das ocorrências	80,19 FP
Módulo 4 – Banco do conhecimento	42,57 FP
Módulo 5 – Administração	41,58 FP
Módulo 6 – Relatórios	165,33 FP
Total SCO	538,56 FP

Através dos totais (por módulos/geral) gerados pela técnica de pontos-por-função e com base nos dados históricos apresentados, conclui-se que o tamanho do projeto (por módulos/geral) em *LOCs* de *Java* (linhas de código de *Java*) é:

Módulo 1 – Cadastro de ocorrências	128,70 FP	5920,20 LOC
Módulo 2 – Workflow das ocorrências	80,19 FP	3688,74 LOC
Módulo 3 – Fechamento das ocorrências	80,19 FP	3688,74 LOC
Módulo 4 – Banco do conhecimento	42,57 FP	1958,22 LOC
Módulo 5 – Administração	41,58 FP	1912,68 LOC
Módulo 6 – Relatórios	165,33 FP	7605,18 LOC
Total SCO	538,56 FP	24773,76 LOC

3. Riscos do Projeto

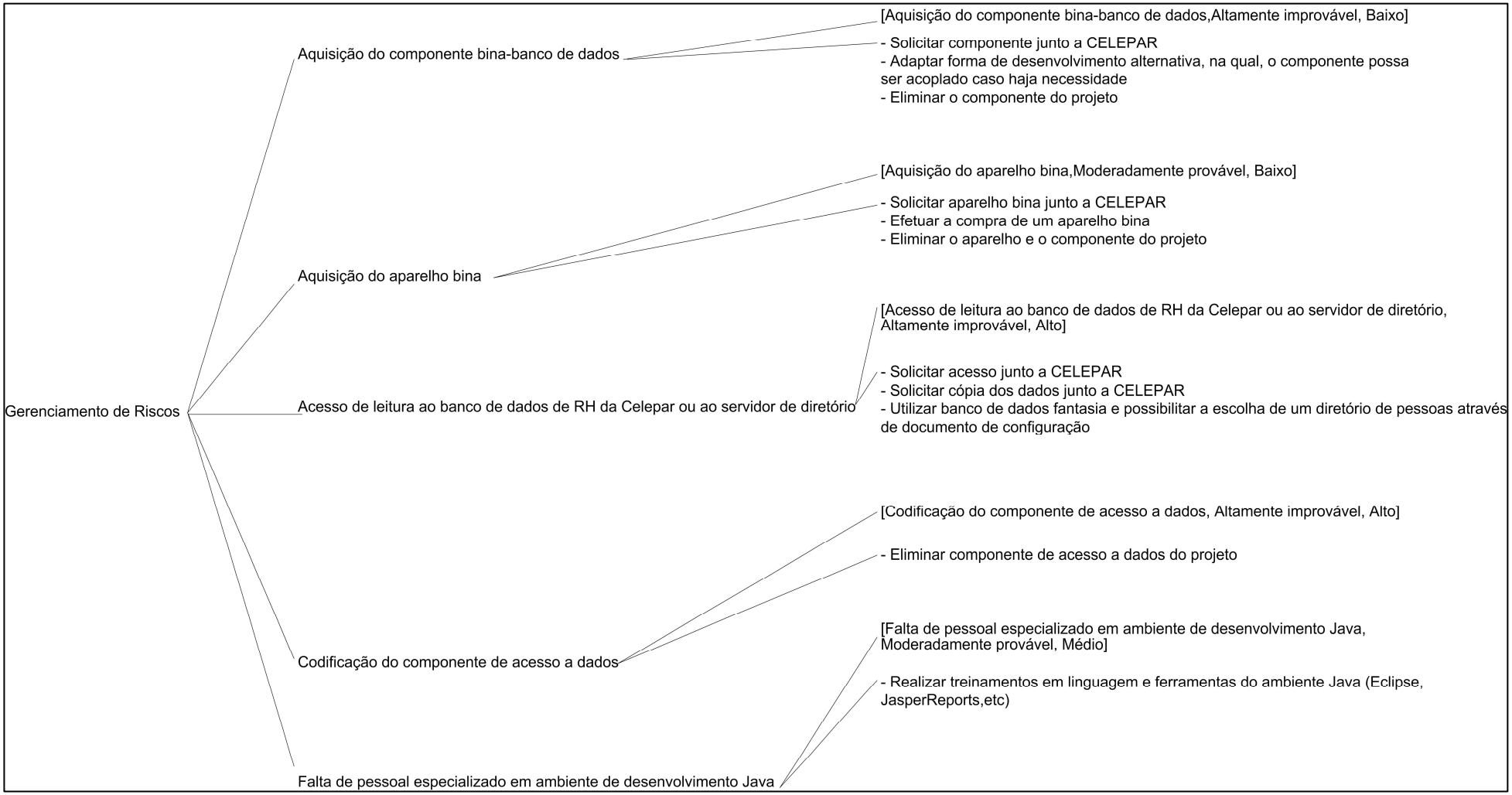
3.1 Análise dos riscos

Foram considerados e estimados os riscos:

	Identificação dos Riscos	Projeção dos Riscos	Impacto no Projeto
Riscos do Projeto	Aquisição do componente bina-banco de dados	Altamente improvável	Baixo
	Aquisição do aparelho bina	Moderadamente provável	Baixo
	Acesso de leitura ao banco de dados de RH da Celepar ou ao servidor de diretório	Altamente improvável	Alto
	Desenvolvimento do banco do conhecimento	Moderadamente provável	Baixo
	Falta de pessoal especializado em ambiente de desenvolvimento Java	Moderadamente provável	Médio
Riscos Técnicos	Codificação do componente de acesso a dados	Altamente improvável	Alto

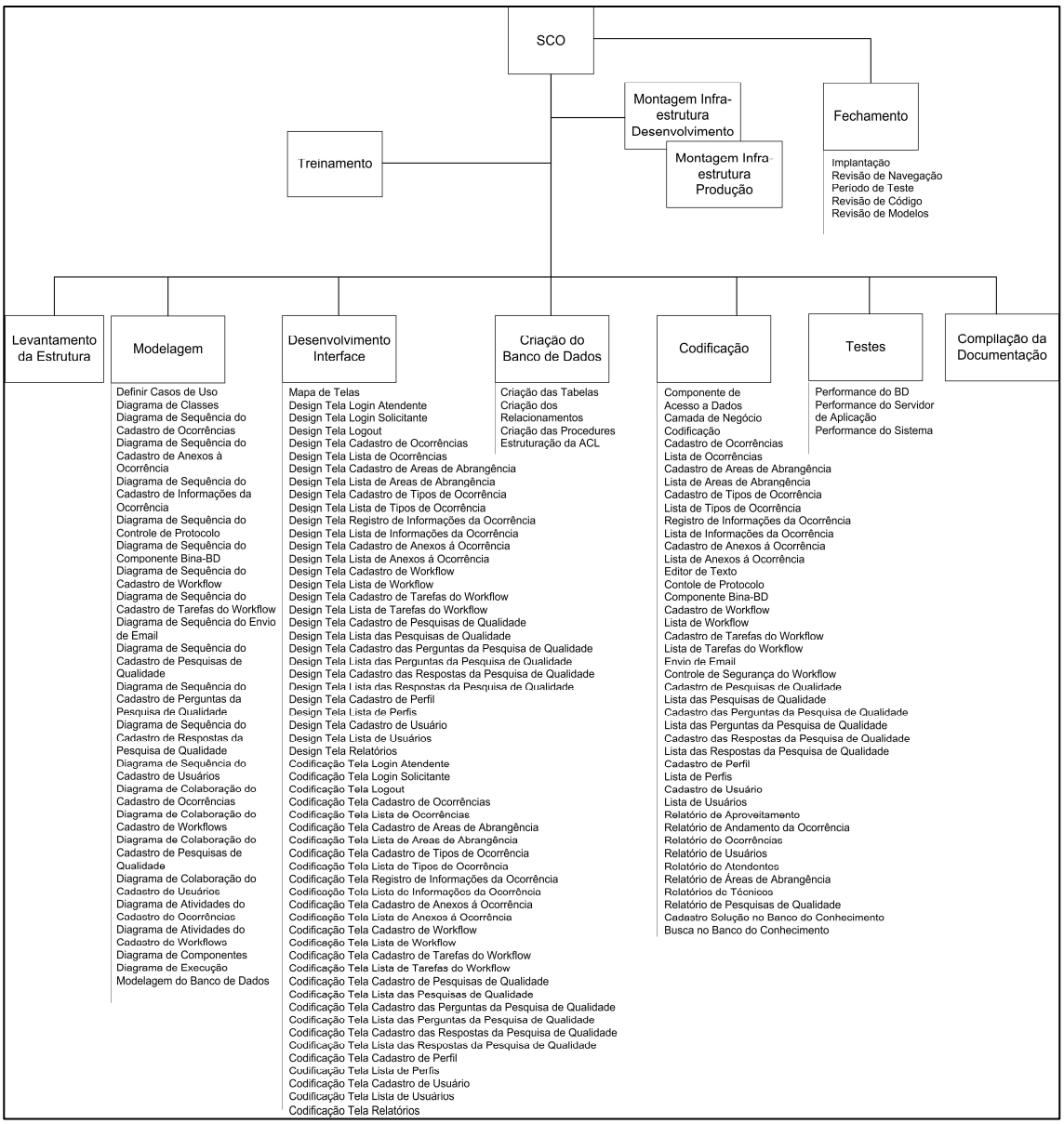
3.2 Administração dos riscos

O gerenciamento dos riscos ocorrerá conforme o diagrama abaixo:



4. Cronograma

4.1 WBS – Work Breakdown Structure



4.2 Rede de Tarefas (Anexo I)

4.3 Gráfico de Gantt (Anexo II)

4.4 Tabela de Recursos

Nome do Recurso	Tipo	Iniciais	Grupo
Fabricio	Trabalho	ASUP	Analista de Suporte
Mirian	Trabalho	WEB	Programador de Interface/Designer
Rodrigo	Trabalho	AJR	Analista Junior
Ramilio	Trabalho	AJR	Analista Junior
Fabricio	Trabalho	AP	Analista Pleno
Mirian	Trabalho	AT	Analista de Treinamento
Ramilio	Trabalho	AN	Analista de Negócio
Fabricio	Trabalho	GP	Gerente de Projetos
Débora	Trabalho	AD	Administrador de Dados
Débora	Trabalho	DBA	Analista de Banco de Dados (DBA)
Mirian	Trabalho	DOC	Documentador
Rodrigo	Trabalho	HOM	Homologador
Rodrigo	Trabalho	IMP	Implantador

5. Recursos do projeto

5.1 Recursos de Pessoal

Serão necessários para a realização deste projeto os profissionais abaixo discriminados:

Funções	Quantidade
Analista de Suporte (Infra-estrutura)	1
Programador de Interface/ Designer	1
Analista Junior	2
Analista Pleno	1
Analista de Treinamento	1
Analista de Negócio	1
Gerente de Projetos	1
Administrador de Dados	1
Analista de Banco de Dados (DBA)	1
Documentador	1
Homologador	1
Implantador	1

5.2 Recursos de hardware e software

Os recursos de software dividem-se em quatro ambientes: desenvolvimento, produção, banco de dados de desenvolvimento e banco de dados de produção.

Produção

Software	Versão	Descrição	Licenças
JasperReport	-	Ferramenta de Relatórios	Free
JBoss Application Server	4.0.5.GA	Servidor de Aplicação	Free
Hibernate Framework	3.0	Framework para persistência de dados e mapeamento objeto-relacional	Free
Struts Framework	1.3.5	Framework MVC	Free
Windows Server	2000 ou superior	Sistema Operacional	1(uma)
Debian	-	Sistema Operacional	Free
PGAdmin	3 v1.6	Administrador do PostgreSQL 8.1	Free
JDBC	-	Middleware de conexão ao banco de dados	Free

Banco de dados de produção

Software	Versão	Descrição	Licenças
PostgreSQL	8.1	Banco de Dados	Free
Windows Server	2000 ou superior	Sistema Operacional	1(uma)
Debian	-	Sistema Operacional	Free

Desenvolvimento

Software	Versão	Descrição	Licenças
JBoss Eclipse IDE	3.2	IDE Java	Free
CVS	2.5.03	Controle de Versões	Free
JasperReport	-	Ferramenta de Relatórios	Free
Macromedia Dreamweaver	8	Design	1(uma)
Macromedia Flash	8	Animação/Design	1(uma)
Adobe Photoshop	CS	Imagem	1(uma)
Corel Photo Suite	12	Imagem	1(uma)
JBoss	4.0.5.GA	Servidor de Aplicaçao	Free
Windows Server	2000 ou superior	Sistema Operacional	1(uma)
Debian	-	Sistema Operacional	Free
Mozilla Firefox	-	Navegador Web	Free
Internet Explorer	3.01 ou superior	Navegador Web	Vinculado a licença windows
PGAdmin	3 v1.4	Administrador do PostgreSQL 8.1	Free
Hibernate Framework	3.0	Framework para persistência de dados e mapeamento objeto-relacional	Free
Struts Framework	1.3.5	Framework MVC	Free
JDBC	-	Middleware de conexão ao banco de dados	Free
MS Project	-	Gestão do Projeto	1(uma)

Banco de dados de desenvolvimento

Software	Versão	Descrição	Licenças
PostgreSQL	8.1	Banco de Dados	Free
Windows Server	2000 ou superior	Sistema Operacional	1(uma)
Debian	-	Sistema Operacional	Free

5.3 Recursos Especiais

Os recursos especiais utilizados serão:

a. Componente para leitura de dados de uma Bina;
b. Conexão com o banco de dados de RH do cliente;
c. Servidor de e-mail;

6. Organização do Pessoal

6.1 Estrutura de equipe

A equipe será estruturada acima das exigências do item 5.1.

Quantidade	Função	Recurso Humano
1	Analista de Suporte (Infra-estrutura)	Fabrício
1	Programador de Interface/ Designer	Mirian
2	Analista Junior	Ramilio, Rodrigo
1	Analista Pleno	Fabrício
1	Analista de Treinamento	Mirian
1	Analista de Negócio	Ramilio
1	Gerente de Projetos	Fabrício
1	Administrador de Dados	Débora
1	Analista de Banco de Dados (DBA)	Débora
1	Documentador	Mirian
1	Homologador	Rodrigo
1	Implantador	Rodrigo

6.2 Relatórios Administrativos

Serão utilizados sempre que eventos de elevada importância aconteçam no projeto e monitoramento.

São eventos de importância:

a. Conclusão de tarefas do cronograma;
b. Conclusão de tarefas críticas;
c. Reuniões ordinárias da equipe de desenvolvimento;
d. Alteração do escopo do projeto;
e. Alteração da Infra-estrutura planejada;
f. Alteração do modelo de negócios planejado;
g. Alteração do modelo de dados planejado;
h. Justificativa de utilização de recursos;

É monitoramento:

a. Monitoramento de riscos;

b. Monitoramento de aproveitamento da equipe;

7. Mecanismos de *tracking* e controle

Os mecanismos de *tracking* e controle serão os relatórios apresentados no item 6.2.

APÊNDICE II – REVISÕES DO PLANO GERAL DE PROJETO

APLICAÇÃO DE *DESIGN PATTERNS* E *LIGHTWEIGHT FRAMEWORKS* NO DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETOS REUTILIZÁVEIS: S.C.O. – SISTEMA DE CONTROLE DE OCORRÊNCIAS

Revisão do Plano de Projeto (v.7.0)

1. Introdução

1.1 Escopo e Propósito do Documento

O presente documento tem o objetivo de apresentar um planejamento para a aplicação de *design patterns* e *lightweight frameworks* no desenvolvimento de um Sistema de Controle de Ocorrências (SCO) que faz o acompanhamento das ocorrências técnicas ou não técnicas, internas ou externas de uma empresa.

O sistema será desenvolvido em parceria com a empresa CELEPAR e funcionará em ambiente w*eb*. A linguagem utilizada será o *Java* e banco de dados *PostgreSql*.

Este documento é constituído pelo detalhamento dos objetivos do projeto, uma análise de risco, cronogramas, estimativas, organização da equipe bem como as principais funções que serão desempenhadas pela ferramenta.

1.2 Escopo do software

O sistema cadastra as ocorrências através do contato de um usuário via telefone ou internet.

No contato via telefone o SCO coletará automaticamente o número de telefone do usuário através de um componente para a leitura de um aparelho identificador de chamada (bina), verificando a possível existência do usuário no banco de dados, e dando procedimento ao cadastro da ocorrência. No contato via internet o sistema irá cadastrar o usuário, caso ele não exista, e a ocorrência.

Após estes cadastros o SCO redirecionará a ocorrência, via e-mail, para o gestor da área de abrangência responsável pela solução desta. O sistema permitirá ao gestor apontar um ou mais técnicos para a análise e resolução da ocorrência.

Uma vez solucionada a ocorrência o SCO cadastrará a solução explorada pelos técnicos. Caso o atendimento tenha ocorrido via telefone o sistema fornecerá aos técnicos responsáveis às informações do solicitante para que este seja informado da solução e o chamado é então finalizado. Caso o atendimento tenha ocorrido via internet o solicitante será informado da solução via e-mail e o chamado será finalizado.

No caso do atendimento telefônico, estando à ocorrência finalizada, o SCO a fará voltar, via e-mail, para o atendente inicial ou para o atendente com menor fluxo de trabalho. Na seqüência o sistema disponibilizará as informações do solicitante para que o atendente entre em contato e cadastre a satisfação do cliente no atendimento.

No caso do atendimento via internet o SCO disponibilizará ao usuário uma área para a avaliação do atendimento.

O solicitante poderá acompanhar o andamento de sua ocorrência a qualquer momento do processo.

1.2 Objetivos

Objetivos do Projeto
A aplicação de *design patterns* e *ligthweight frameworks* no desenvolvimento de *softwares* orientados a objetos reutilizáveis.
Documentar as ocorrências internas/externas.
Assegurar a melhor assistência técnica à ocorrência.
Manter o solicitante informado sobre a situação de sua ocorrência.
Acumular o conhecimento realizado no atendimento.
Melhoria na qualidade do atendimento
Melhoria do aproveitamento dos recursos humanos responsáveis pela resolução das ocorrências

1.2.1 Funções Principais

Módulo 1: Cadastro de ocorrências:

2. Controle de ocorrências;

Módulo 2: *Workflow* das ocorrências:

2. Controle de *workflow*;

Módulo 3: Fechamento das ocorrências:

2. Controle de avaliações de atendimento;

Módulo 4: Banco do conhecimento (Adicional)

2. Controle de documentos do banco de conhecimento;

Módulo 5: Administração

1. Controle de Usuários;

Módulo 6: Relatórios

2. Controle de relatórios;

2. Estimativas de Projeto

2.1 Dados históricos usados nas estimativas

Os dados históricos que servirão de base para realizar as estimativas foram extraídos da revista técnica *Software Development* de outubro de 2000. A revista apresenta a seguinte análise:

Linguagem de Programação	LOC por FP
C++	53
Cobol	107
Delphi 5	18
HTML 4	14
Visual Basic 6	24
SQL	13
Java	46

Estes dados serão utilizados para conversão de pontos-por-função (F.P. - *Functions Points*) em linhas de código (LOC – *Lines of Code*) nas análises de estimativa.

2.2 Técnicas de estimativas

Foram utilizadas as técnicas de pontos-por-função (F.P. - *Functions Points*) e LOC (*Lines of code*).

A métrica de pontos-por-função foi primariamente proposta por Albrecht [ALB79] e é baseada no fato de que o tamanho dos sistemas não depende das tecnologias empregadas, mas sim da especificação ou “funcionabilidade”.

Já a técnica de LOC – linhas de código – consiste em estimar o tamanho do software em linhas de código a serem programadas. Embora seja uma das métricas que mais se aproximam do real esforço a ser realizado para a confecção de um projeto, é altamente dependente da linguagem de programação utilizada.

Desta forma, ambas as técnica serão empregadas para gerar as estimativas deste projeto.

2.3 Estimativas

Estimativas em pontos-por-função das principais funções do SCO:

Módulo 1: Cadastro de ocorrências

1. Controle de ocorrências;

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**5**	X	3	4	6	**6**	=	30
Número de saídas do usuário	**0**	X	4	6	7	**4**	=	0
Número de consultas do usuário	**5**	X	3	4	6	**3**	=	15
Número de arquivos	**5**	X	7	10	15	**15**	=	75
Número de interfaces externas	**1**	X	5	7	10	**10**	=	10
						Total	=	130

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação

$FP = Total \times [0,65 + 0,01 \times SOMA(F_i)]$ = 128,7 FP

Módulo 2: *Workflow* das ocorrências;

1. Controle de *workflow*

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**3**	X	3	4	6	**6**	=	18
Número de saídas do usuário	**0**	X	4	6	7	**4**	=	0
Número de consultas do usuário	**3**	X	3	4	6	**6**	=	18
Número de arquivos	**3**	X	7	10	15	**15**	=	45
Número de interfaces externas	**0**	X	5	7	10	**0**	=	0
						Total	=	81

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação		
$FP = Total \times [0,65 + 0,01 \times SOMA(F_i)]$	=	80,19 FP

Módulo 3: Fechamento das ocorrências

1. Controle de avaliações de atendimento;

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**3**	X	3	4	6	**6**	=	18
Número de saídas do usuário	**0**	X	4	6	7	**7**	=	0
Número de consultas do usuário	**3**	X	3	4	6	**6**	=	18
Número de arquivos	**3**	X	7	10	15	**15**	=	45
Número de interfaces externas	**0**	X	5	7	10		=	0
						Total	=	81

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação		
$FP = Total \times [0,65 + 0,01 \times SOMA(F_i)]$	=	80,19 FP

Módulo 4: Banco do conhecimento (Adicional)

1. Controle de documentos do banco de conhecimento;

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**1**	X	3	4	6	**6**	=	6
Número de saídas do usuário	**0**	X	4	6	7	**4**	=	0
Número de consultas do usuário	**4**	X	3	4	6	**3**	=	12
Número de arquivos	**1**	X	7	10	15	**15**	=	15
Número de interfaces externas	**1**	X	5	7	10	**10**	=	10
						Total	=	43

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação		
$FP = Total \times [0,65 + 0,01 \times SOMA(F_i)]$	=	42,57 FP

Módulo 5: Administração

1.Controle de usuários;

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**2**	X	3	4	6	**3**	=	6
Número de saídas do usuário	**0**	X	4	5	7	**4**	=	0
Número de consultas do usuário	**2**	X	3	4	6	**3**	=	6
Número de arquivos	**2**	X	7	10	15	**10**	=	20
Número de interfaces externas	**1**	X	5	7	10	**10**	=	10
						Total	=	42

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação		
FP = Total x [0,65 + 0,01 x SOMA(F_i)]	=	41,58 FP

Módulo 6: Relatórios

1.Controle de relatórios;

			Fator de Ponderação					
Parâmetro de medida	**Contagem**		Simples	Médio	Complexo	**Escolhido**		
Número de entradas do usuário	**1**	X	3	4	6	**6**	=	6
Número de saídas do usuário	**8**	X	4	6	7	**7**	=	56
Número de consultas do usuário	**8**	X	3	4	6	**6**	=	48
Número de arquivos	**0**	X	7	10	15	**7**	=	0
Número de interfaces externas	**2**	X	5	7	10	**10**	=	20
						Total	=	130

Ajustes de Complexidade	**Grau**
Comunicação de Dados	**4**
Funções Distribuídas	**4**
Performance	**0**
Configuração do equipamento	**1**
Volume de transações	**3**
Entrada de dados on-line	**5**
Interface com o usuário	**3**
Atualização on-line	**1**
Processamento complexo	**0**
Reusabilidade	**5**
Facilidade de implantação	**0**
Facilidade operacional	**1**
Multiplos locais	**3**
Facilidade de mudanças (Felixibilidade)	**4**
Soma	**34**

Aplicação da relação		
$FP = Total \times [0,65 + 0,01 \times SOMA(F_i)]$	=	128,7 FP

Totalização (por módulos/geral)

Módulo 1 – Cadastro de ocorrências	128,70 FP
Módulo 2 – Workflow das ocorrências	80,19 FP
Módulo 3 – Fechamento das ocorrências	80,19 FP
Módulo 4 – Banco do conhecimento	42,57 FP
Módulo 5 – Administração	41,58 FP
Módulo 6 – Relatórios	165,33 FP
Total SCO	538,56 FP

Através dos totais (por módulos/geral) gerados pela técnica de pontos-por-função e com base nos dados históricos apresentados, conclui-se que o tamanho do projeto (por módulos/geral) em *LOCs* de *Java* (linhas de código de *Java*) é:

Módulo 1 – Cadastro de ocorrências	128,70 FP	5920,20 LOC
Módulo 2 – Workflow das ocorrências	80,19 FP	3688,74 LOC
Módulo 3 – Fechamento das ocorrências	80,19 FP	3688,74 LOC
Módulo 4 – Banco do conhecimento	42,57 FP	1958,22 LOC
Módulo 5 – Administração	41,58 FP	1912,68 LOC
Módulo 6 – Relatórios	165,33 FP	7605,18 LOC
Total SCO	538,56 FP	24773,76 LOC

3. Riscos do Projeto

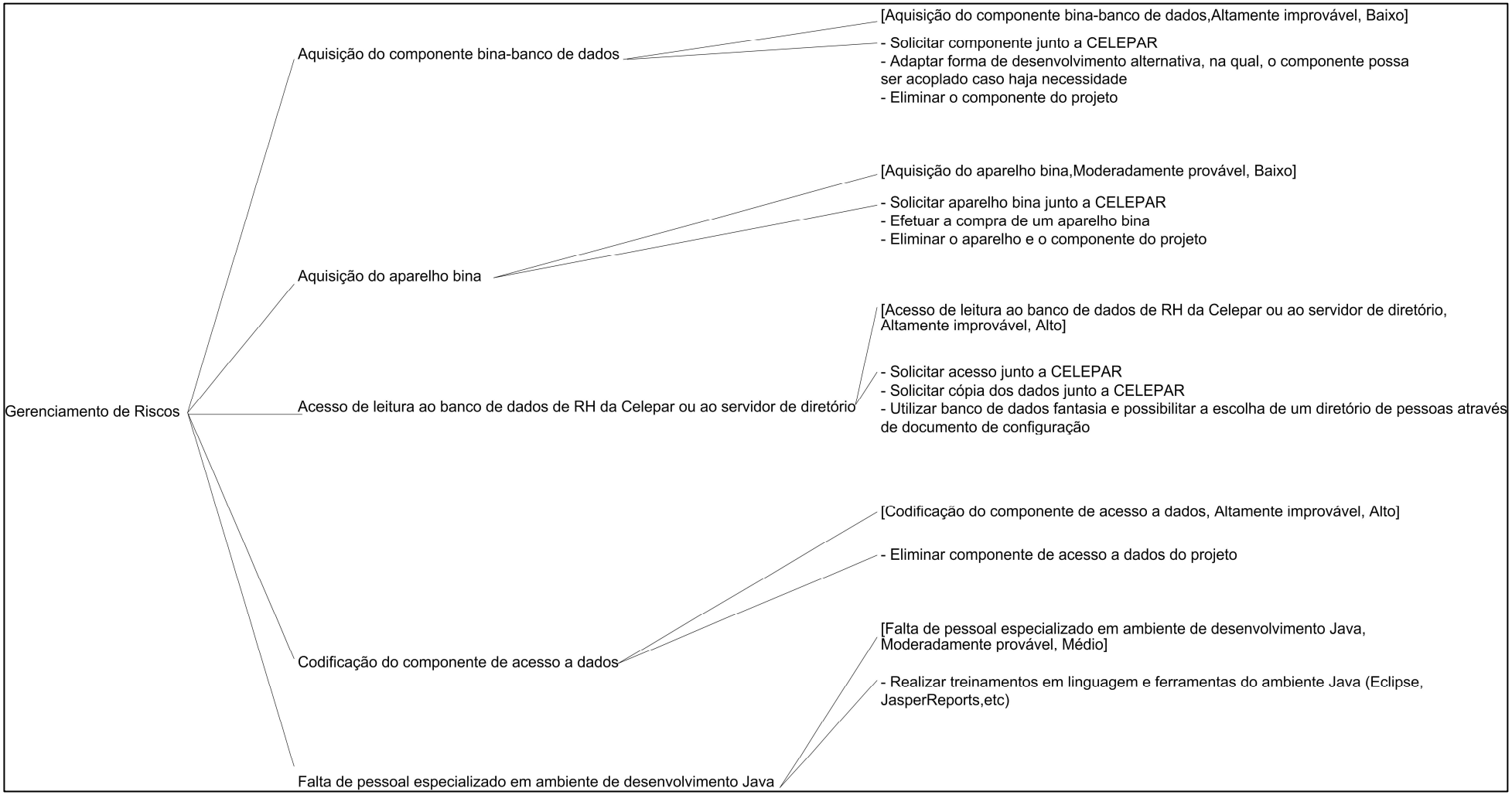
3.1 Análise dos riscos

Foram considerados e estimados os riscos:

	Identificação dos Riscos	Projeção dos Riscos	Impacto no Projeto
Riscos do Projeto	Aquisição do componente bina-banco de dados	Altamente improvável	Baixo
	Aquisição do aparelho bina	Moderadamente provável	Baixo
	Acesso de leitura ao banco de dados de RH da Celepar ou ao servidor de diretório	Altamente improvável	Alto
	Desenvolvimento do banco do conhecimento	Moderadamente provável	Baixo
	Falta de pessoal especializado em ambiente de desenvolvimento Java	Moderadamente provável	Médio
Riscos Técnicos	Codificação do componente de acesso a dados	Altamente improvável	Alto

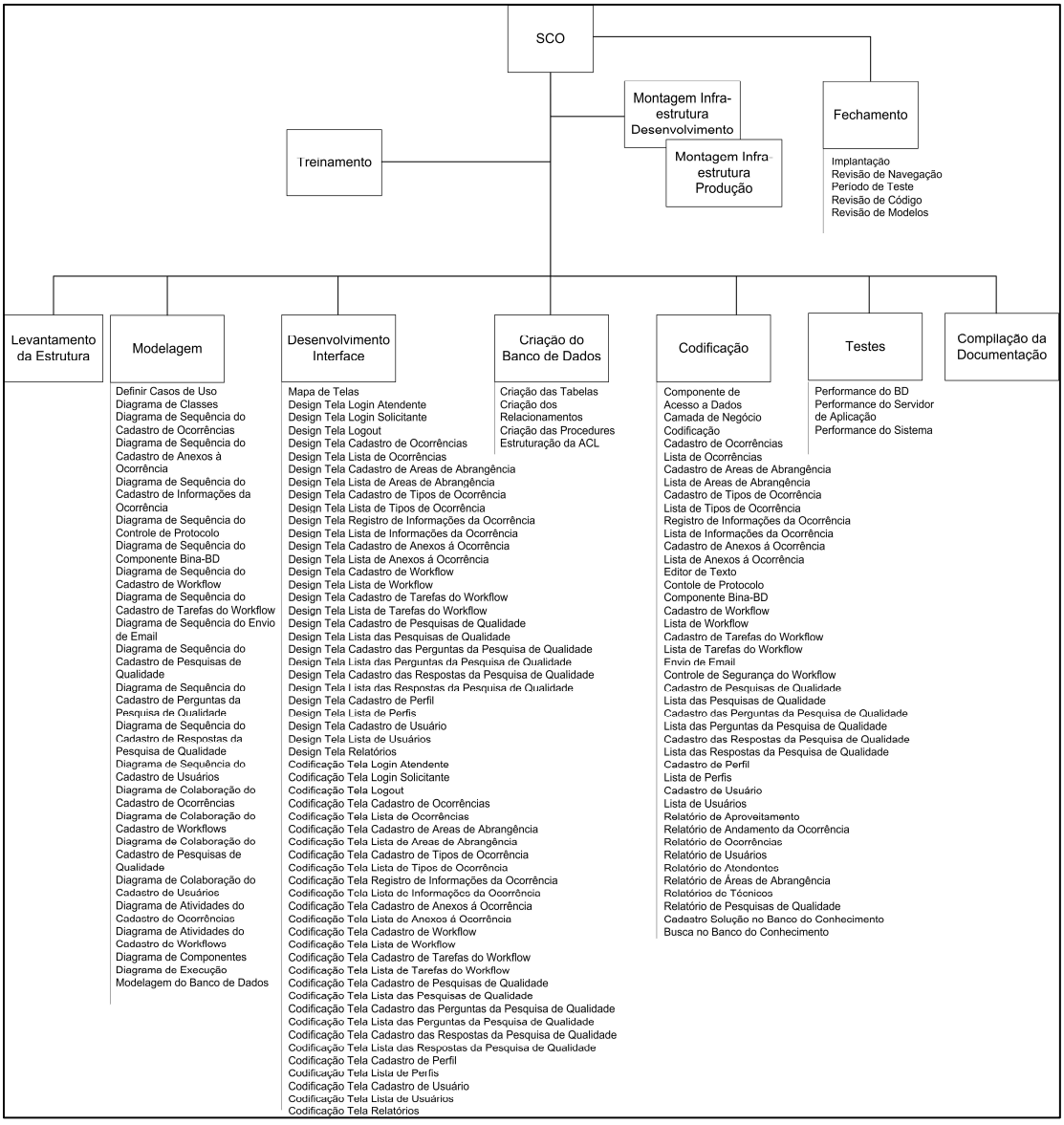
3.2 Administração dos riscos

O gerenciamento dos riscos ocorrerá conforme o diagrama abaixo:



4. Cronograma

4.1 WBS – Work Breakdown Structure



4.2 Rede de Tarefas (Anexo I)

4.3 Gráfico de Gantt (Anexo II)

4.4 Tabela de Recursos

Nome do Recurso	Tipo	Iniciais	Grupo
Fabricio	Trabalho	ASUP	Analista de Suporte
Mirian	Trabalho	WEB	Programador de Interface/Designer
Rodrigo	Trabalho	AJR	Analista Junior
Ramilio	Trabalho	AJR	Analista Junior
Fabricio	Trabalho	AP	Analista Pleno
Mirian	Trabalho	AT	Analista de Treinamento
Ramilio	Trabalho	AN	Analista de Negócio
Fabricio	Trabalho	GP	Gerente de Projetos
Débora	Trabalho	AD	Administrador de Dados
Débora	Trabalho	DBA	Analista de Banco de Dados (DBA)
Mirian	Trabalho	DOC	Documentador
Rodrigo	Trabalho	HOM	Homologador
Rodrigo	Trabalho	IMP	Implantador

5. Recursos do projeto

5.1 Recursos de Pessoal

Serão necessários para a realização deste projeto os profissionais abaixo discriminados:

Funções	Quantidade
Analista de Suporte (Infra-estrutura)	1
Programador de Interface/ Designer	1
Analista Junior	2
Analista Pleno	1
Analista de Treinamento	1
Analista de Negócio	1
Gerente de Projetos	1
Administrador de Dados	1
Analista de Banco de Dados (DBA)	1
Documentador	1
Homologador	1
Implantador	1

5.2 Recursos de hardware e software

Os recursos de software dividem-se em quatro ambientes: desenvolvimento, produção, banco de dados de desenvolvimento e banco de dados de produção.

Produção

Software	Versão	Descrição	Licenças
JasperReport	-	Ferramenta de Relatórios	Free
JBoss Application Server	4.0.5.GA	Servidor de Aplicação	Free
Hibernate Framework	3.0	Framework para persistência de dados e mapeamento objeto-relacional	Free
Struts Framework	1.3.5	Framework MVC	Free
Windows Server	2000 ou superior	Sistema Operacional	1(uma)
Debian	-	Sistema Operacional	Free
PGAdmin	3 v1.6	Administrador do PostgreSQL 8.1	Free
JDBC	-	Middleware de conexão ao banco de dados	Free

Banco de dados de produção

Software	Versão	Descrição	Licenças
PostgreSQL	8.1	Banco de Dados	Free
Windows Server	2000 ou superior	Sistema Operacional	1(uma)
Debian	-	Sistema Operacional	Free

Desenvolvimento

Software	Versão	Descrição	Licenças
JBoss Eclipse IDE	3.2	IDE Java	Free
CVS	2.5.03	Controle de Versões	Free
JasperReport	-	Ferramenta de Relatórios	Free
Macromedia Dreamweaver	8	Design	1(uma)
Macromedia Flash	8	Animação/Design	1(uma)
Adobe Photoshop	CS	Imagem	1(uma)
Corel Photo Suite	12	Imagem	1(uma)
JBoss	4.0.5.GA	Servidor de Aplicaçao	Free
Windows Server	2000 ou superior	Sistema Operacional	1(uma)
Debian	-	Sistema Operacional	Free
Mozilla Firefox	-	Navegador Web	Free
Internet Explorer	3.01 ou superior	Navegador Web	Vinculado a licença windows
PGAdmin	3 v1.4	Administrador do PostgreSQL 8.1	Free
Hibernate Framework	3.0	Framework para persistência de dados e mapeamento objeto-relacional	Free
Struts Framework	1.3.5	Framework MVC	Free
JDBC	-	Middleware de conexão ao banco de dados	Free
MS Project	-	Gestão do Projeto	1(uma)

Banco de dados de desenvolvimento

Software	Versão	Descrição	Licenças
PostgreSQL	8.1	Banco de Dados	Free
Windows Server	2000 ou superior	Sistema Operacional	1(uma)
Debian	-	Sistema Operacional	Free

5.3 Recursos Especiais

Os recursos especiais utilizados serão:

d. Componente para leitura de dados de uma Bina;
e. Conexão com o banco de dados de RH do cliente;
f. Servidor de e-mail;

6. Organização do Pessoal

6.1 Estrutura de equipe

A equipe será estruturada acima das exigências do item 5.1.

Quantidade	Função	Recurso Humano
1	Analista de Suporte (Infra-estrutura)	Fabrício
1	Programador de Interface/ Designer	Fabrício
2	Analista Junior	Fabrício
1	Analista Pleno	Fabrício
1	Analista de Treinamento	Rodrigo
1	Analista de Negócio	Fabrício
1	Gerente de Projetos	Fabrício
1	Administrador de Dados	Rodrigo
1	Analista de Banco de Dados (DBA)	Rodrigo
1	Documentador	Rodrigo
1	Homologador	Rodrigo
1	Implantador	Fabrício

6.2 Relatórios Administrativos

Serão utilizados sempre que eventos de elevada importância aconteçam no projeto e monitoramento.

São eventos de importância:

i. Conclusão de tarefas do cronograma;
j. Conclusão de tarefas críticas;
k. Reuniões ordinárias da equipe de desenvolvimento;
l. Alteração do escopo do projeto;
m. Alteração da Infra-estrutura planejada;

n. Alteração do modelo de negócios planejado;
o. Alteração do modelo de dados planejado;
p. Justificativa de utilização de recursos;

É monitoramento:

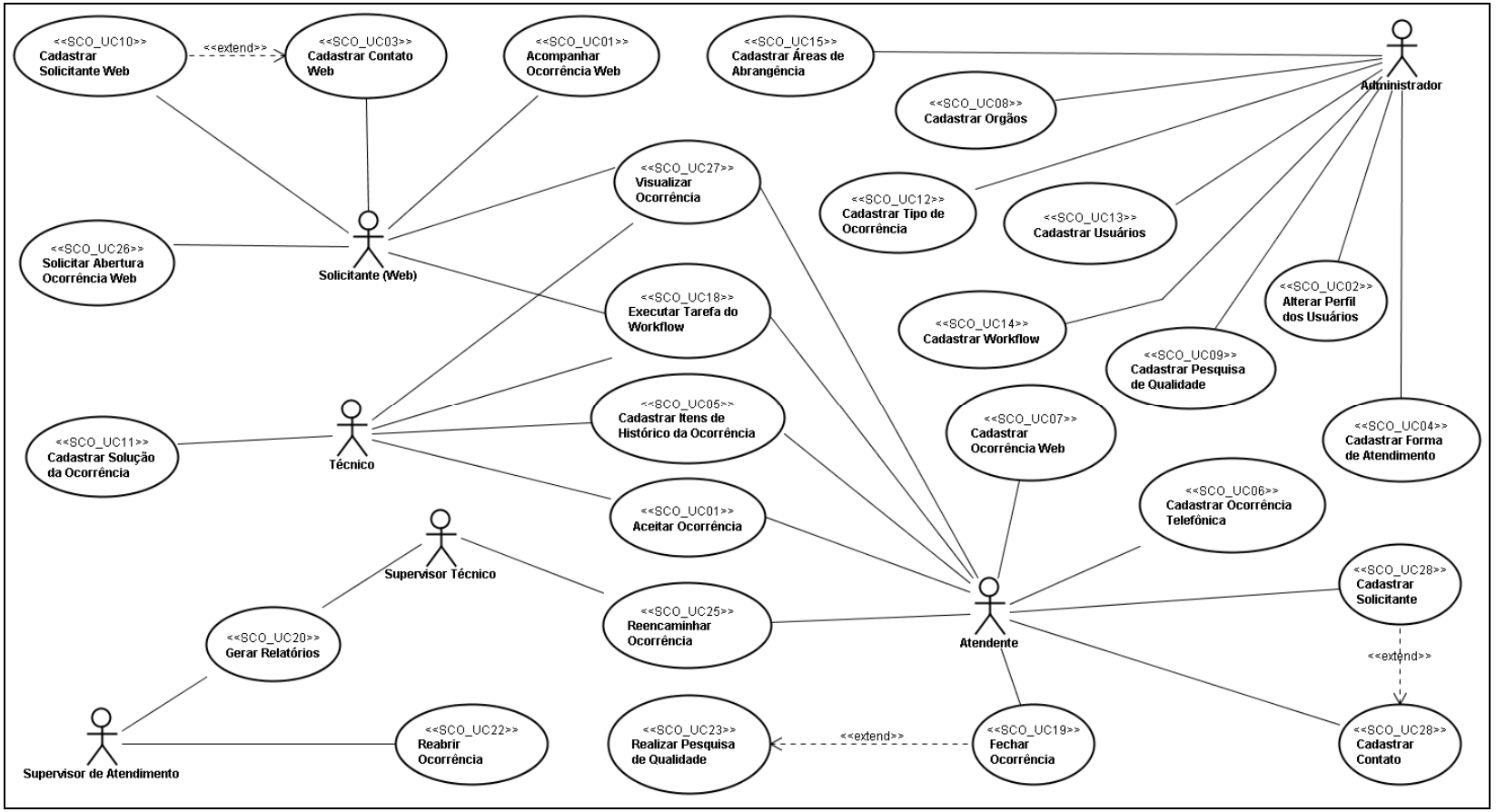
c. Monitoramento de riscos;
d. Monitoramento de aproveitamento da equipe;

7. Mecanismos de *tracking* e controle

Os mecanismos de *tracking* e controle serão os relatórios apresentados no item 6.2.

APÊNDICE II – MODELAGEM ORIENTADA A OBJETOS (UML)

DIAGRAMA DE CASOS DE USO


<<SCO_UC10>>
Cadastrar Solicitante Web
<<extend>>
<<SCO_UC03>>
Cadastrar Contato Web
<<SCO_UC01>>
Acompanhar Ocorrência Web
<<SCO_UC15>>
Cadastrar Áreas de Abrangência
Administrador
<<SCO_UC08>>
Cadastrar Orgãos
<<SCO_UC27>>
Visualizar Ocorrência
<<SCO_UC12>>
Cadastrar Tipo de Ocorrência
<<SCO_UC13>>
Cadastrar Usuários
<<SCO_UC26>>
Solicitar Abertura Ocorrência Web
Solicitante (Web)
<<SCO_UC18>>
Executar Tarefa do Workflow
<<SCO_UC14>>
Cadastrar Workflow
<<SCO_UC02>>
Alterar Perfil dos Usuários
<<SCO_UC09>>
Cadastrar Pesquisa de Qualidade
<<SCO_UC05>>
Cadastrar Itens de Histórico da Ocorrência
<<SCO_UC07>>
Cadastrar Ocorrência Web
<<SCO_UC04>>
Cadastrar Forma de Atendimento
<<SCO_UC11>>
Cadastrar Solução da Ocorrência
Técnico
<<SCO_UC01>>
Aceitar Ocorrência
<<SCO_UC06>>
Cadastrar Ocorrência Telefônica
Supervisor Técnico
<<SCO_UC28>>
Cadastrar Solicitante
<<SCO_UC25>>
Reencaminhar Ocorrência
Atendente
<<SCO_UC20>>
Gerar Relatórios
<<extend>>
Supervisor de Atendimento
<<SCO_UC22>>
Reabrir Ocorrência
<<SCO_UC23>>
Realizar Pesquisa de Qualidade
<<extend>>
<<SCO_UC19>>
Fechar Ocorrência
<<SCO_UC28>>
Cadastrar Contato

DESCRIÇÃO DOS CASOS DE USO

Caso de uso: Cadastrar Solicitante.

Ator: Atendente.

Descrição Resumida:

O atendente identifica a necessidade do cadastro de um solicitante, insere os dados no sistema e efetiva o cadastramento.

Pré-requisitos:

O atendente deve estar autenticado no sistema com o perfil de acesso atendente.

Fluxo de Eventos:

Fluxo Principal: O atendente identifica a necessidade de cadastrar um solicitante. Para isso ele reúne os dados deste e após todos os campos obrigatórios estarem preenchidos efetiva o cadastramento.

Cenários

Cenário Principal

Durante um atendimento telefônico o atendente identifica que o solicitante não foi previamente cadastrado. Para efetuar o atendimento o atendente deve primeiro cadastrar o solicitante para depois abrir a ocorrência.

Sendo assim, para efetivar o cadastro do solicitante o atendente informa ao sistema os dados do solicitante coletados durante o atendimento telefônico.

Caso de uso: Cadastrar Contato.

Ator: Atendente.

Descrição Resumida:

O atendente necessita cadastrar um contato para um determinado solicitante.

Pré-requisitos:

O atendente deve estar autenticado no sistema com o perfil de acesso atendente.

O solicitante que terá um novo contato adicionado deve estar previamente cadastrado no sistema.

Fluxo de Eventos:

Fluxo Principal: O atendente recebe uma ligação e verifica que será necessário cadastrar um novo contato para um solicitante já cadastrado. Ele acessa o cadastro do solicitante, seleciona a operação para adicionar um novo contato e salva as informações inseridas.

Cenários

Cenário Principal

Durante um atendimento telefônico o atendente identifica uma situação na qual deve cadastrar um novo contato para um solicitante previamente cadastrado. O atendente insere no sistema os dados do novo contato e efetiva o cadastro.

Caso de uso: Cadastrar Ocorrência Telefônica.

Ator: Atendente.

Descrição Resumida:

O atendente recebe uma solicitação para registrar a abertura de uma ocorrência. Após fazer cadastrá-la a ocorrência é automaticamente encaminhada à área técnica responsável pelo atendimento.

Pré-requisitos:

O atendente deve estar autenticado no sistema com o perfil de acesso atendente.

Fluxo de Eventos:

Fluxo Principal: O atendente recebe uma ligação e verifica se o solicitante está cadastrado (A1). Após o solicitante ser identificado no sistema, o atendente registra qual é a descrição da ocorrência. Cada ocorrência, quando criada, adquire um número de protocolo que é informado ao solicitante para que este possa acompanhar o andamento.

Por fim a ocorrência à automaticamente encaminhada à área responsável pelo atendimento através do preenchimento das informações referentes ao workflow.

Fluxos Alternativos:

A1: Cadastrar Solicitante:

Em uma situação na qual o solicitante não está cadastrado o atendente deve executar o caso de uso: *Cadastrar Solicitante* antes do registro da ocorrência.

Cenários

Cenário Principal

Durante o atendimento telefônico a um solicitante atendente identifica a necessidade de cadastrar uma nova ocorrência para atender a solicitação. Ele consulta na aplicação o solicitante, para certificar-se de que este já esteja cadastrado. Após isso insere a descrição detalhada da ocorrência, informando qual o tipo de ocorrência, o que permitirá que esta seja encaminhada para a área responsável pelo atendimento.

Cenário Secundário

Durante um atendimento telefônico o atendente recebe a solicitação para cadastrar uma nova ocorrência., mas verifica que o solicitante não possui cadastro. O atendente solicita os dados do solicitante e efetua o cadastro para que na seqüência a ocorrência seja aberta.

Caso de uso: Cadastrar Ocorrência Web.

Ator: Atendente.

Descrição Resumida:

O atendente recebe, via e-mail, uma solicitação para registrar uma ocorrência. Ele verifica se os dados estão preenchidos corretamente e cadastra a ocorrência. Após isso é enviado um e-mail de confirmação para o solicitante.

Pré-requisitos:

O atendente deve estar autenticado no sistema com o perfil de acesso atendente.

Fluxo de Eventos:

Fluxo Principal: O atendente recebe via e-mail uma ocorrência que deverá ser registrada. Ele executa o caso de uso *Visualizar Ocorrência*, verifica se os dados estão preenchidos corretamente e executa a operação para cadastrar a ocorrência. Na seqüência um e-mail de confirmação é enviado para o solicitante.

Cenários

Cenário Principal

Um solicitante deseja cadastrar uma ocorrência para sanar uma dúvida sobre a alíquota de exportação do feijão. Como ele possui internet, acessa a aplicação e realiza o auto-atendimento. Após preencher todos os dados obrigatórios ele solicita a abertura da ocorrência. Esta é enviada para um atendente, aleatoriamente, que irá confirmar os dados e registrar a ocorrência. Na seqüência é enviado um e-mail de confirmação para o solicitante.

Caso de uso: Visualizar Ocorrência.

Ator: Atendente.

Descrição Resumida:

O atendente deseja localizar uma ocorrência no sistema e visualizar os dados.

Pré-requisitos:

O atendente deve estar autenticado no sistema com o perfil de acesso atendente.

A ocorrência a ser localizada deve estar previamente cadastrada no sistema.

Fluxo de Eventos:

Fluxo Principal: O atendente deseja localizar uma ocorrência e para isso deve informar o número de protocolo da ocorrência ou realizar uma busca através dos dados do solicitante. Após encontrá-la a ocorrência e carregada e seus dados exibidos.

Cenários

Cenário Principal

Durante um atendimento telefônico o atendente identifica a necessidade de encontrar uma ocorrência já existente e visualizar seus dados. Ele solicita, ao solicitante o número do protocolo da ocorrência ou algum dado pessoal para realizar a consulta. De posse dessas informações o atendente realiza uma consulta no sistema e após localizada a ocorrência é carregada e seus dados exibidos em tela.

Caso de uso: Cadastrar Itens de Histórico da Ocorrência

Ator: Atendente.

Descrição Resumida:

O atendente identifica a necessidade de cadastrar informações de andamento de uma ocorrência. Ele visualiza a ocorrência no sistema e adiciona um novo item no histórico.

Pré-requisitos:

O atendente deve estar autenticado no sistema com o perfil de acesso atendente.

A ocorrência deve estar previamente cadastrada no sistema.

Fluxo de Eventos:

Fluxo Principal: O atendente identifica a necessidade de cadastrar informações de andamento de uma ocorrência previamente cadastrada. Ele deve executar o caso de uso *Visualizar Ocorrência*, acessar a operação adicionar item de histórico, digitar os dados e cadastrar o item.

Cenários

Cenário Principal

Durante um atendimento telefônico o atendente recebe a solicitação para adicionar novas informações a uma ocorrência previamente cadastrada. Para localizar a ocorrência o atendente executa o caso de uso *Visualizar Ocorrência*, acessar o histórico desta e selecionar a operação para cadastrar um item de histórico.

Caso de uso: Aceitar Ocorrência.

Ator: Técnico.

Descrição Resumida:

O técnico recebe uma ocorrência encaminhada pelo sistema, e após verificar se o atendimento da mesma é de sua competência, a aceita para resolução.

Pré-requisitos:

O técnico deve estar autenticado no sistema com o perfil de acesso técnico.

Fluxo de Eventos:

Fluxo Principal: O técnico recebe uma ocorrência encaminhada pelo sistema e verifica se o atendimento desta é de sua competência (A1). Após a verificação ele executa a operação de aceite.

Fluxos Alternativos:

A1: Caso o técnico receba uma ocorrência que não é de sua competência e ele deve executar a ação de negar ocorrência, para que esta retorne ao elo anterior do workflow e possa seguir o encaminhamento correto.

Cenários

Cenário Principal:

O técnico recebe um e-mail alertando que existe uma ocorrência encaminhada para ele que está aguardando ser aceita. O técnico verifica se o conteúdo da ocorrência é de sua competência. Após a verificação ele executa a operação de aceite da ocorrência.

Cenário Secundário

O técnico recebe um e-mail alertando que existe uma ocorrência encaminhada para ele que está aguardando ser aceita. O técnico verifica se o conteúdo da ocorrência é de sua competência. Após verificar descobre que a ocorrência não é de sua competência mas sim de um técnico de outra área de abrangência. Sendo assim, ele nega a ocorrência e esta retorna para quem o designou a tarefa.

Caso de uso: Executar Tarefa do Workflow.

Ator: Técnico.

Descrição Resumida:

O técnico recebe uma ocorrência encaminhada pelo sistema e executa a tarefa pertinente.

Pré-requisitos:

O técnico deve estar autenticado no sistema com o perfil de acesso técnico.

Fluxo de Eventos:

Fluxo Principal: O técnico recebe a ocorrência, faz os procedimentos descritos no caso de uso Aceitar Ocorrência e executa a tarefa de sua atribuição conforme o workflow pertinente ao tipo da solicitação.

Cenários

Cenário Principal

O técnico recebe uma ocorrência encaminhada pelo sistema e executa o caso de uso Aceitar Ocorrência. Após aceitar a ocorrência ele executa a tarefa de sua atribuição, conforme o workflow pertinente ao tipo da solicitação, inserindo as informações necessárias para a finalização da tarefa.

Caso de uso: Reencaminhar Ocorrência.

Ator: Supervisor técnico.

Descrição Resumida:

O supervisor técnico localiza uma ocorrência e altera o técnico qual ela será encaminhada.

Pré-requisitos:

O supervisor técnico deve estar autenticado no sistema com o perfil de acesso supervisor técnico.

A ocorrência que será reencaminhada deve estar cadastrada anteriormente no sistema.

Fluxo de Eventos:

Fluxo Principal: O supervisor técnico identifica a necessidade de reencaminhar uma ocorrência aceita por um técnico para outro técnico. Para isso ele deve executar o caso de uso *Visualizar Ocorrência* e depois reencaminhar a ocorrência ao técnico desejado.

Cenários

Cenário Principal

O supervisor técnico identifica a necessidade de reencaminhar uma ocorrência aceita por um técnico entrou em período de férias para outro técnico que poderá atender a solicitação. O supervisor técnico consulta a ocorrência executando o caso de uso *Visualizar Ocorrência* e informa qual técnico receberá a ocorrência reencaminhada.

Caso de uso: Fechar Ocorrência.

Ator: Atendente.

Descrição Resumida:

O atendente recebe uma ocorrência já solucionada e que deve ser fechada.

Pré-requisitos:

O atendente deve estar autenticado no sistema com o perfil de acesso atendente.

A ocorrência deve ter sido solucionada.

Fluxo de Eventos:

Fluxo Principal: O atendente recebe via e-mail uma ocorrência solucionada. Ele deve localizar a ocorrência, executando o caso de uso *Visualizar Ocorrência*, e efetuar o fechamento (A1).

Fluxos Alternativos:

A1: Aleatoriamente algumas ocorrências são escolhidas para que seja realizada uma pesquisa de qualidade sobre o processo de atendimento. Neste caso o atendente deverá executar o caso de uso *Realizar Pesquisa de Qualidade*.

Cenários

Cenário Principal

O atendente recebe um alerta via e-mail informando que uma ocorrência sobre o valor de impostos para exportação de soja foi solucionada e deve ser encerrada. Inicialmente ele deve localizar a ocorrência, executando o caso de uso *Visualizar Ocorrência*, e efetuar o fechamento.

Cenário Secundário

Após efetuar o fechamento da ocorrência o atendente, é sorteado pelo sistema para realizar uma pesquisa de qualidade no atendimento junto ao solicitante. Ele executa todos os passos do caso de uso *Realizar Pesquisa de Qualidade*.

Caso de uso: Realizar Pesquisa de Qualidade.

Ator: Atendente.

Descrição Resumida:

O atendente deve realizar uma pesquisa de qualidade sobre o atendimento de uma ocorrência sorteada pelo sistema. Para isso, o atendente entra em contato com o solicitante e realiza a pesquisa.

Pré-requisitos:

O atendente deve estar autenticado no sistema com o perfil de acesso atendente.

A ocorrência deve ter sido atendida solucionada.

Fluxo de Eventos:

Fluxo Principal: O atendente é sorteado pelo sistema para realizar uma pesquisa de qualidade sobre o atendimento de uma ocorrência. Ele localiza a ocorrência sorteada pelo sistema e executa o caso de uso *Visualizar Ocorrência*, para exibir seus detalhes. Após isso o atendente entra em contato com o solicitante e realiza a pesquisa, cadastrando no sistema as respostas.

Cenários

Cenário Principal

O sistema sorteia uma ocorrência para que seja realizada uma pesquisa de qualidade sobre o atendimento e envia uma notificação a um atendente. Este localiza a ocorrência, executando o caso de uso *Visualizar Ocorrência*, entra em contato com o solicitante e realiza a pesquisa, fazendo as perguntas ao solicitante, cadastrando no sistema as respostas.

Caso de uso: Cadastrar Áreas de Abrangência

Ator: Administrador

Descrição Resumida:

O administrador recebe uma solicitação externa ao sistema para cadastrar uma nova área de abrangência. Ele acessa o sistema e efetua o cadastramento.

Pré-requisitos:

O administrador deve estar autenticado no sistema com o perfil de acesso administrador.

Fluxo de Eventos:

Fluxo Principal: O administrador recebe uma solicitação externa ao sistema para que seja cadastrada uma nova área de abrangência. Ele efetua o cadastramento inserindo no sistema os dados obrigatórios referentes ao cadastramento de uma nova área de abrangência.

Cenários

Cenário Principal

O administrador recebe uma solicitação externa ao sistema para cadastrar uma nova área de abrangência responsável por dúvidas de plantão fiscal especificas para alíquotas de imposto sobre produtos que serão exportados, separando desta forma das demais dúvidas de cunho fiscal. O administrador efetua o cadastramento inserindo no sistema todos dados obrigatórios referentes ao cadastramento de uma nova área de abrangência.

Caso de uso: Cadastrar Órgãos

Ator: Administrador

Descrição Resumida:

O administrador recebe uma solicitação externa ao sistema para cadastrar um novo órgão. Ele acessa o sistema, insere os dados do órgão e efetiva o cadastro.

Pré-requisitos:

O administrador deve estar autenticado no sistema com o perfil de acesso administrador.

Fluxo de Eventos:

Fluxo Principal: O administrador recebe uma solicitação externa ao sistema para cadastrar um novo órgão. Para isso ele deve acessar o sistema e inserir os dados obrigatórios para o cadastro e por fim efetivar o cadastro.

Cenários

Cenário Principal

O administrador recebe uma solicitação da gerência para que seja cadastrado um novo órgão, a Secretária do Estado da Saúde do Paraná, que começará a utilizar o sistema para receber denuncias referentes a possíveis focos do mosquito transmissor da dengue. Para isso o administrador deve acessar o sistema e inserir os dados obrigatórios para cadastrar um novo órgão e efetivar o cadastro.

Caso de uso: Cadastrar Tipo de Ocorrência

Ator: Administrador

Descrição Resumida:

O administrador recebe uma solicitação externa ao sistema para cadastrar no sistema um novo tipo de ocorrência. Para isso ele acessa o sistema, insere os dados do tipo de ocorrência e efetiva o cadastro.

Pré-requisitos:

O administrador deve estar autenticado no sistema com o perfil de acesso administrador.

Fluxo de Eventos:

Fluxo Principal: O administrador recebe uma solicitação externa ao sistema para cadastrar um novo tipo de ocorrência. Para isso ele deve acessar o sistema e inserir os dados obrigatórios para o cadastro de um novo tipo de ocorrência e efetivar o cadastro.

Cenários

Cenário Principal

O administrador recebe uma solicitação da Secretária do Estado da Saúde do Paraná para que seja cadastrado um novo tipo de ocorrência chamada “Consulta”, com o intuito de que através do sistema possa ser recebido o agendamento de consultas médicas nos hospitais da rede estadual. Para isso o administrador deve acessar o sistema e inserir os dados obrigatórios para cadastrar um novo tipo de ocorrência e efetivar o cadastro.

Caso de uso: Cadastrar Usuários

Ator: Administrador

Descrição Resumida:

O administrador recebe uma solicitação externa ao sistema para cadastrar um novo usuário. Para isso ele acessa o sistema, insere os dados obrigatórios do usuário e efetiva o cadastramento.

Pré-requisitos:

O administrador deve estar autenticado no sistema com o perfil de acesso administrador.

Fluxo de Eventos:

Fluxo Principal: O administrador recebe uma solicitação externa ao sistema para cadastrar um novo usuário. Para isso ele deve acessar o sistema e inserir os dados obrigatórios para o cadastro efetivá-lo.

Cenários

Cenário Principal

O administrador recebe uma solicitação externa ao sistema para que seja cadastrado um novo usuário, pois um novo funcionário passará a utilizá-lo. Para isso o administrador deve acessar o sistema e inserir os dados obrigatórios para o cadastro efetiva-lo.

Caso de uso: Alterar Perfil dos Usuários

Ator: Administrador

Descrição Resumida:

O administrador recebe uma solicitação externa ao sistema para alterar o perfil de acesso de um usuário. Para isso ele acessa o sistema, informa qual é o usuário que terá seu perfil de acesso alterado e efetua a alteração.

Pré-requisitos:

O administrador deve estar autenticado no sistema com o perfil de acesso administrador.

Fluxo de Eventos:

Fluxo Principal: O administrador recebe uma solicitação externa ao sistema para alterar o perfil de acesso de um usuário. Para isso ele acessa o sistema, informa qual é o usuário que terá seu perfil de acesso alterado, seleciona qual é o novo perfil de acesso e efetiva a alteração.

Cenários

Cenário Principal

O administrador recebe uma solicitação para que seja alterado o perfil de acesso de um funcionário (usuário do sistema) que foi promovido de atendente para supervisor de atendimento. Para isso o administrador acessa o sistema, informa qual é o usuário que terá seu perfil de acesso alterado, seleciona qual é o novo perfil de acesso e efetiva a alteração.

Caso de uso: Cadastrar Forma de Atendimento

Ator: Administrador

Descrição Resumida:

O administrador recebe uma solicitação externa ao sistema para cadastrar uma nova forma de atendimento. Para isso ele acessa o sistema, informa os dados da nova forma de atendimento e efetua o cadastro.

Pré-requisitos:

Fluxo Principal: O administrador deve estar autenticado no sistema com o perfil de acesso administrador.

Fluxo de Eventos:

O administrador recebe uma solicitação externa ao sistema para cadastrar uma nova forma de atendimento. Para isso ele acessa o sistema, informa qual os dados da nova forma de atendimento e efetua o cadastro.

Cenários

Cenário Principal

O administrador recebe uma solicitação para que seja cadastrada uma nova forma de atendimento, o atendimento via chat no site da Secretária da Fazenda Estadual. Para isso o administrador deve acessar o sistema e inserir os dados obrigatórios para cadastrar uma nova forma de atendimento e efetiva o cadastro.

Caso de uso: Cadastrar Pesquisa de Qualidade

Ator: Administrador

Descrição Resumida:

O administrador recebe uma solicitação externa ao sistema para cadastrar uma nova pesquisa de qualidade. Para isso ele acessa o sistema, informa os dados da nova pesquisa de qualidade e efetua o cadastro.

Pré-requisitos:

O administrador deve estar autenticado no sistema com o perfil de acesso administrador.

Fluxo de Eventos:

Fluxo Principal: O administrador recebe uma solicitação externa ao sistema para cadastrar uma nova pesquisa de qualidade. Para isso ele deve informar ao sistema os dados da pesquisa, quais as perguntas e suas respectivas respostas e na seqüência efetuar o cadastro.

Cenários

Cenário Principal

O administrador recebe uma solicitação para criação de uma nova pesquisa de qualidade, com o intuito de permitir que o supervisor da área técnica responsável pela resolução de dúvidas fiscais possa analisar a qualidade do atendimento. O administrador deve informar ao sistema os dados da pesquisa, quais as perguntas e suas respectivas respostas e na seqüência efetuar o cadastro.

Caso de uso: Cadastrar Workflow

Ator: Administrador

Descrição Resumida:

O administrador recebe uma solicitação externa ao sistema para cadastrar um novo workflow para um tipo de ocorrência. Para isso ele acessa o sistema, seleciona o tipo de ocorrência que receberá o novo workflow, informa os dados deste e efetua o cadastro.

Pré-requisitos:

O administrador deve estar autenticado no sistema com o perfil de acesso administrador.

O tipo de ocorrência deve estar previamente cadastrado.

Fluxo de Eventos:

Fluxo Principal: O administrador recebe uma solicitação externa ao sistema para cadastrar um novo workflow para determinado tipo de ocorrência. Para isso ele acessa o sistema, seleciona o tipo de ocorrência que receberá o novo workflow, informa os dados deste, cadastra suas tarefas e aponta os respectivos responsáveis.

Cenários

Cenário Principal

O administrador recebe uma solicitação para criação de um workflow que atenda a demanda de um novo tipo de ocorrência: a denuncia de maus tratos de animais. Para isso ele acessa o sistema, seleciona o tipo de ocorrência que receberá o novo workflow, informa os dados deste, cadastra suas respectivas tarefas (como atendimento, encaminhamento para a área responsável, resolução da ocorrência e encerramento) e aponta os respectivos responsáveis.

Caso de uso: Solicitar Abertura Ocorrência Web

Ator: Solicitante

Descrição Resumida:

O solicitante, através da web, acessa o sistema, visualiza seu cadastro pessoal, descreve as informações referentes à sua solicitação e registra a ocorrência.

Pré-requisitos:

O solicitante deve estar previamente cadastrado.

Fluxo de Eventos:

Fluxo Principal: O solicitante, através da web, acessa o sistema, realiza uma consulta para visualizar seu cadastro pessoal (A1), descreve as informações referentes à sua solicitação e registra a ocorrência. Após o registro da ocorrência o solicitante aguarda que sua ocorrência seja aceita pelo atendente e posteriormente atendida.

Fluxos Alternativos:

A1: Caso o solicitante não consiga localizar seu cadastro pessoal ele deverá executar o caso de uso *Cadastrar Solicitante Web* antes do registro da ocorrência.

Cenários

Cenário Principal

O solicitante possui uma dúvida referente à alíquota de imposto ICMS que deve ser recolhida em operações de venda de feijão para outras unidades da federação. Ele acessa o sistema, digita um dado pessoal para buscar seu cadastro pessoal, descreve a dúvida no sistema e efetua a solicitação para abertura de uma nova ocorrência. Após registrá-la o solicitante aguarda a confirmação via e-mail do aceite do atendente para que a solicitação seja encaminhada para solução.

Cenário Secundário

O solicitante possui uma dúvida referente à alíquota de imposto ICMS que deve ser recolhida em operações de venda de feijão para outras unidades da federação. Ele acessa o sistema, digita um dado pessoal para buscar seu cadastro pessoal, porém não o encontra. Ele se recorda de que nunca foi cadastrado no sistema. Sendo assim ele acessa a operação novo cadastro e

informa os dados pessoais exigidos para que na seqüência possa registrar a ocorrência.

Caso de uso: Acompanhar Ocorrência Web

Ator: Solicitante.

Descrição Resumida:

O solicitante, através da web, acessa o sistema para acompanhar o andamento do atendimento de uma ocorrência previamente cadastrada. O sistema disponibiliza para ele a visualização do status e do histórico da ocorrência.

Pré-condições:

O solicitante deve estar autenticado no sistema com o perfil de acesso solicitante.

A ocorrência deve estar previamente cadastrada no sistema.

Fluxo de Eventos:

Fluxo Principal: O solicitante deseja visualizar uma ocorrência e, através da web, acessa o sistema. Ele deve informar o número de protocolo da ocorrência ou selecioná-la entre as ocorrências ligadas ao seu cadastro pessoal. Após isso são exibidos em tela os dados resumidos da ocorrência juntamente com seu andamento.

Cenários

Cenário Principal

O solicitante deseja acompanhar o andamento do atendimento de uma ocorrência previamente cadastrada. Ele acessa o sistema e informa o número de protocolo da ocorrência ou a seleciona entre as ocorrências ligadas a seu cadastro pessoal. Após isso são exibidos em tela os dados resumidos da ocorrência juntamente com o andamento.

Caso de uso: Cadastrar Solicitante web.

Ator: Solicitante.

Descrição Resumida:

O solicitante, através da web, acessa o sistema para realizar seu cadastro.

Pré-condições: Nenhuma.

Fluxo de Eventos:

Fluxo Principal: O solicitante, através da web, acessa o sistema para realizar seu cadastro. Ele preenche um formulário informando seus dados pessoais e o envia. Após isso ele aguarda a confirmação de seu cadastro via e-mail.

Cenários

Cenário Principal

O solicitante, através da web, acessa o sistema para obter a informação sobre qual é o imposto para exportação de soja, mas para registrar a sua dúvida ele deve possuir um cadastro pessoal no sistema. O solicitante preenche um formulário informando seus dados e solicita o cadastro. Está solicitação é recebida por um atendente que verifica se os dados estão informados corretamente e efetua a confirmação do cadastro do solicitante, que recebe um e-mail de notificação.

Caso de uso: Cadastrar Contato Web

Ator: Solicitante.

Descrição Resumida:

O solicitante deseja cadastrar uma pessoa no sistema para que ela também seja contato com o atendimento.

Pré-condições: O solicitante deve estar previamente cadastrado e autenticado no sistema com o perfil de acesso solicitante.

Fluxo de Eventos:

Fluxo Principal: O solicitante, através da web, acessa o sistema para cadastrar um contato. Para isso ele deve preencher um formulário informando os dados do contato e efetuar o cadastro.

Cenários

Cenário Principal

O solicitante abriu uma ocorrência para esclarecer dúvidas sobre parcelamento de dívidas com o Estado, porém terá que se ausentar por alguns dias. Para que seja possível ao técnico informar a solução da ocorrência, o solicitante uma vez ausente, cadastra um contato.

Caso de uso: Reabrir Ocorrência.

Ator: Supervisor de Atendimento.

Descrição Resumida:

O supervisor técnico identifica a necessidade de reabrir uma ocorrência já encerrada.

Pré-condições: O supervisor técnico deve estar autenticado no sistema com o perfil de acesso supervisor técnico.

Fluxo de Eventos:

Fluxo Principal: O supervisor acessa o sistema, executa o caso de uso Visualizar Ocorrência e efetua a operação reabrir ocorrência.

Cenários

Cenário Principal

O atendente finalizou uma ocorrência referente a uma dúvida de plantão fiscal solucionada pela área técnica. Porem o solicitante entrou em contato informando que não recebeu o retorno do atendimento. Sendo assim o atendente solicita ao supervisor técnico que reabra a ocorrência para que ela seja reencaminhada para o técnico responsável, afim de realizar de forma satisfatória o atendimento.

Caso de uso: Gerar Relatórios.

Ator: Supervisor de Atendimento.

Descrição Resumida:

O supervisor de atendimento emite os relatórios relacionados ao atendimento.

Pré-condições: O supervisor de atendimento deve estar autenticado no sistema com o perfil de acesso supervisor de atendimento.

Fluxo de Eventos:

Fluxo Principal: O supervisor acessa o sistema e seleciona qual relatório deseja emitir.

Cenários

Cenário Principal

O supervisor de atendimento deseja verificar com que freqüência os atendentes estão realizando as pesquisas de qualidade no momento do encerramento das ocorrências. Para isso ele acessa o sistema, seleciona a opção relatórios e escolhe o relatório de pesquisa de qualidade.

Caso de uso: Cadastrar Solução da Ocorrência.

Ator: Técnico.

Descrição Resumida:

O técnico visualiza a ocorrência e cadastra a solução.

Pré-condições: O técnico deve estar autenticado no sistema com o perfil de acesso Técnico.

Fluxo de Eventos:

Fluxo Principal: O técnico recebe uma ocorrência que deverá ser solucionada. Após verificar que a ocorrência é pertinente a sua área de abrangência a aceita, cadastra a devida solução e a encaminha para o fechamento.

Cenários

Cenário Principal

O técnico recebe uma ocorrência referente ao plantão fiscal a qual o solicitante possui dúvidas sobre como proceder quando forem extraviadas notas fiscais. O técnico entra em contato com o solicitante, visualiza a ocorrência e após instruir o solicitante a consultar no regulamento do ICMS, o artigo 574, cadastra a solução.

DIAGRAMA DE CLASSES

DIAGRAMA DE CLASSES DA CAMADA DE PERSISTÊNCIA

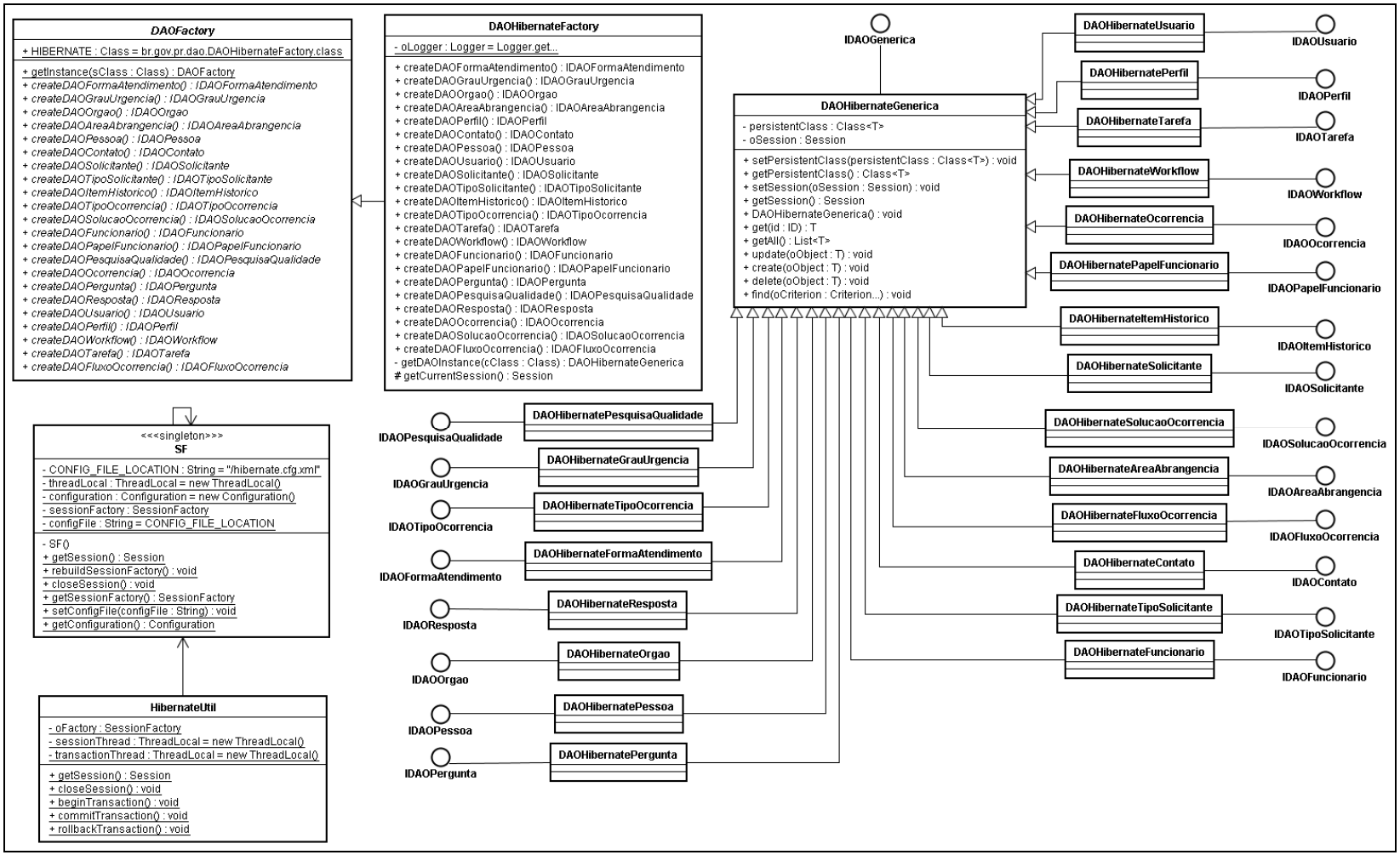


DIAGRAMA DE CLASSES DA CAMADA DE MODELO (FAÇADES)

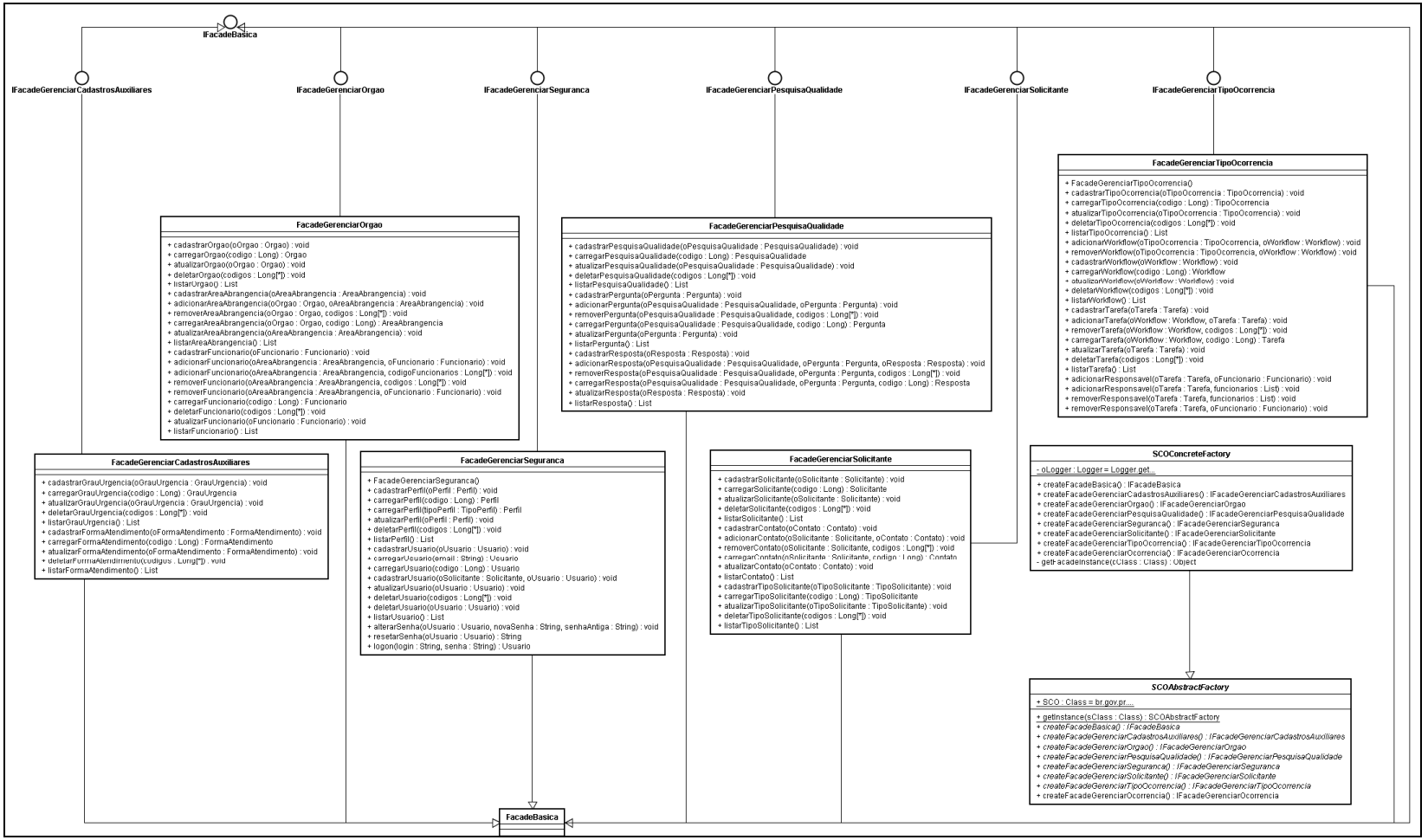
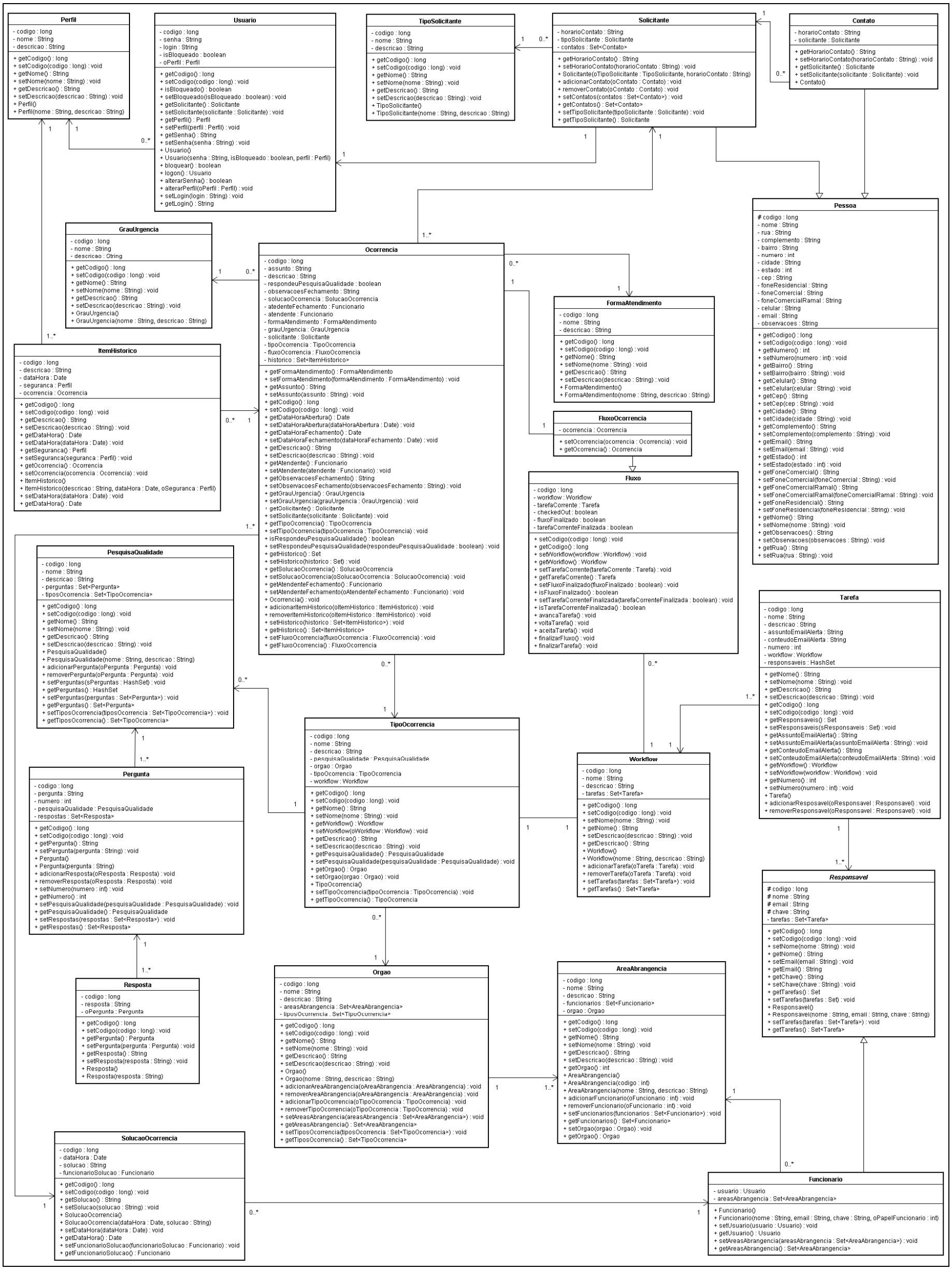
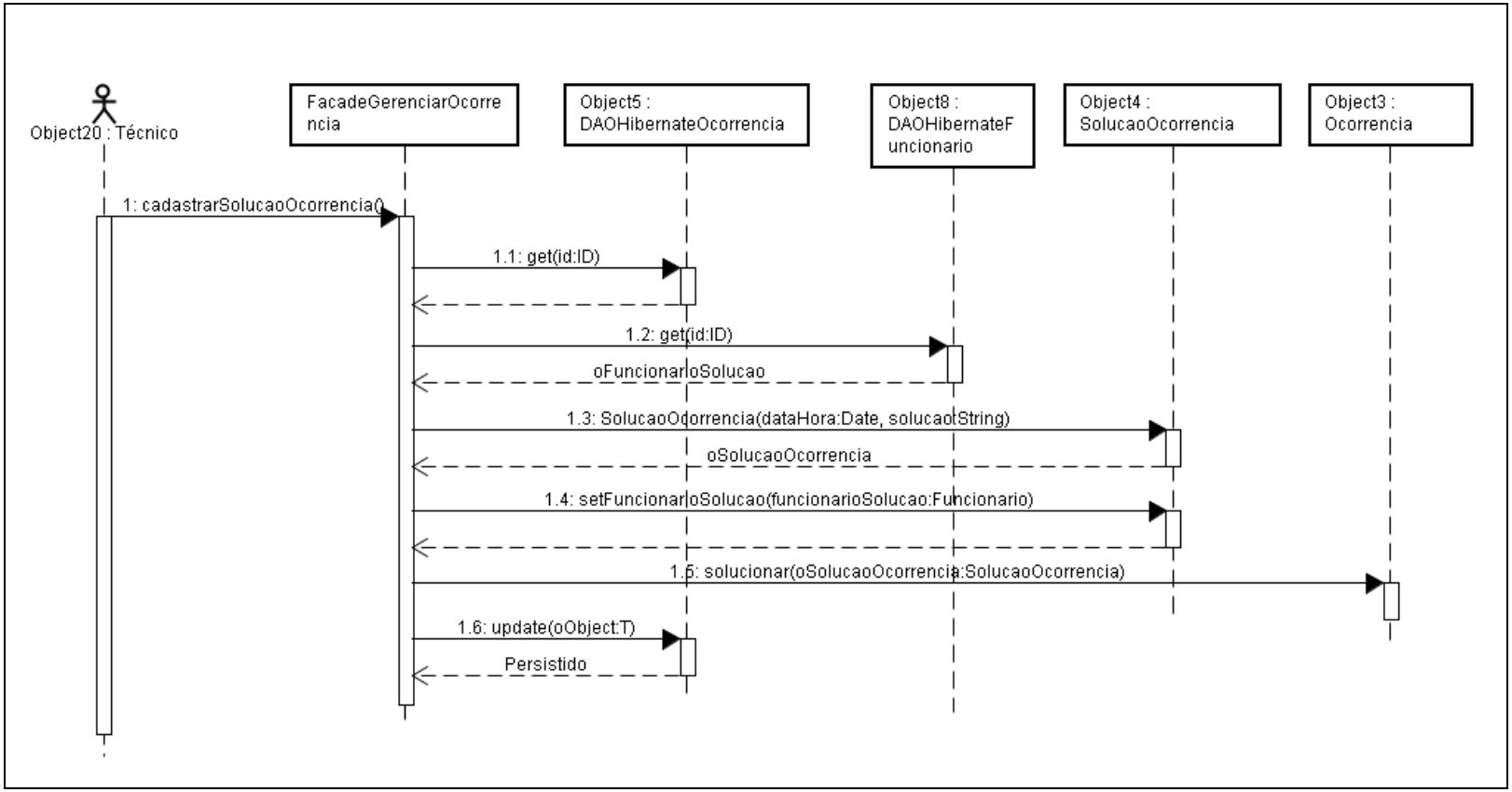


DIAGRAMA DE CLASSES DA CAMADA DE MODELO (OBJETOS DE NEGÓCIO)

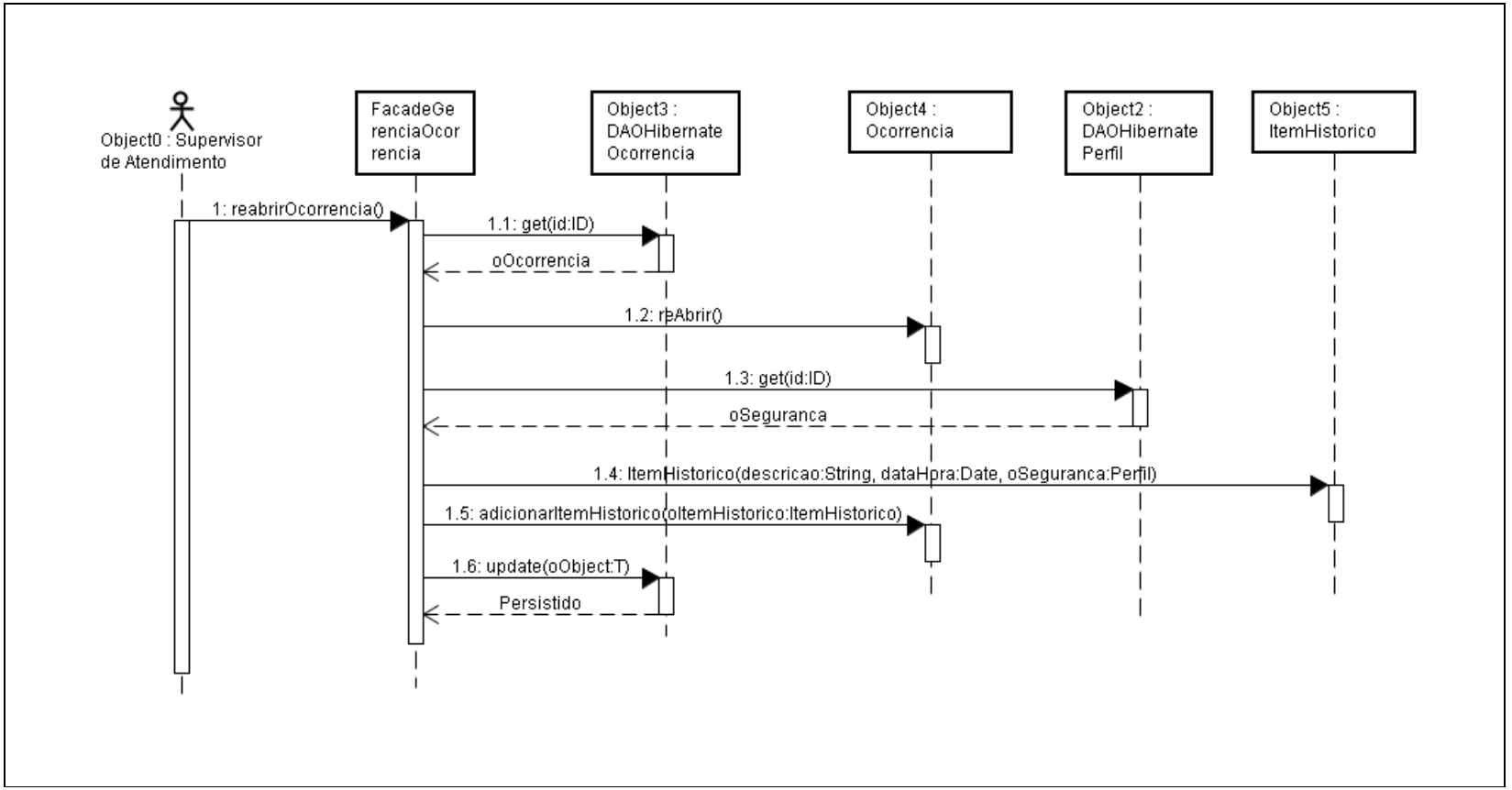


DIAGRAMAS DE SEQUÊNCIA

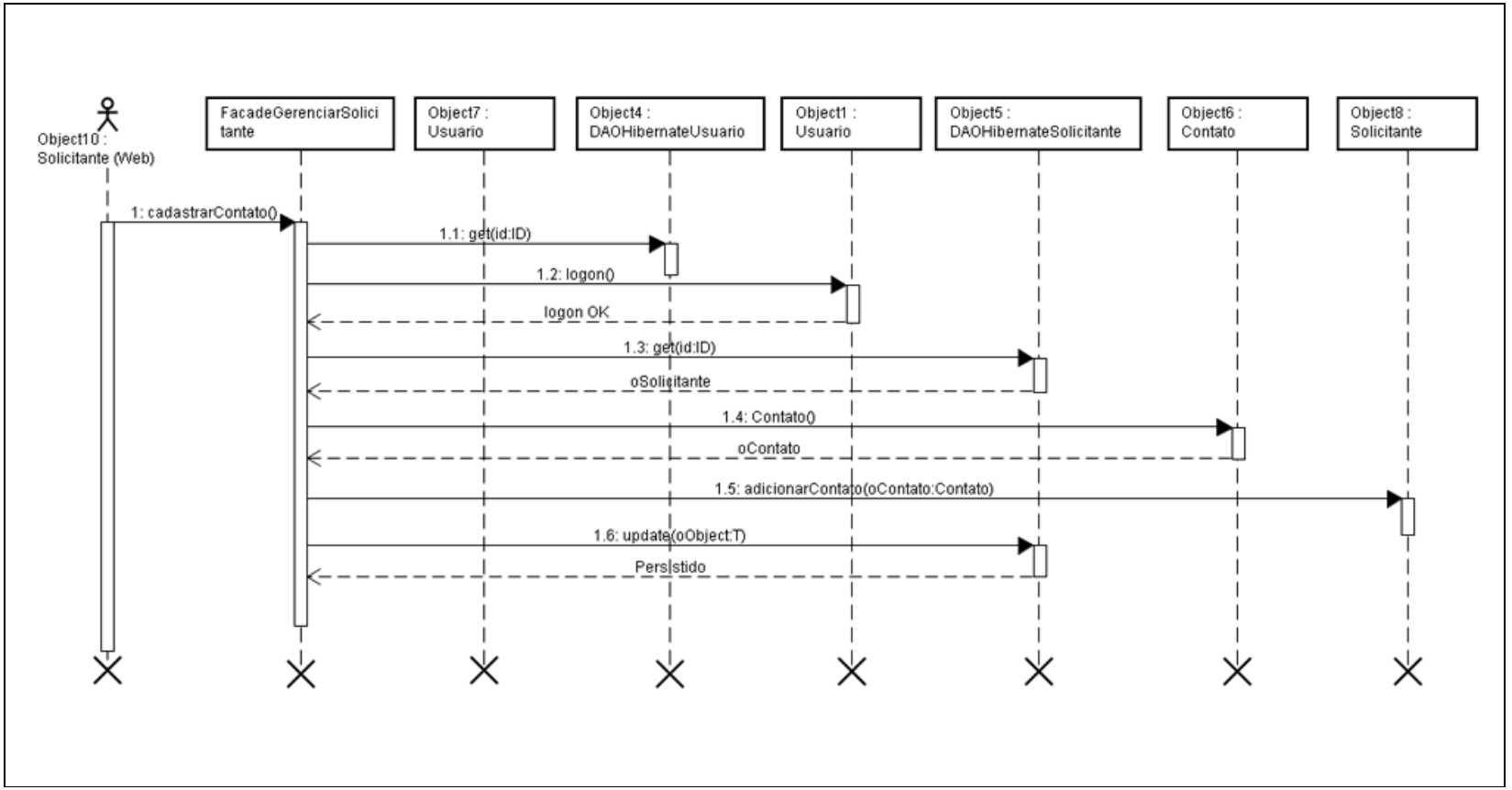
CADASTRAR SOLUÇÃO DA OCORRÊNCIA



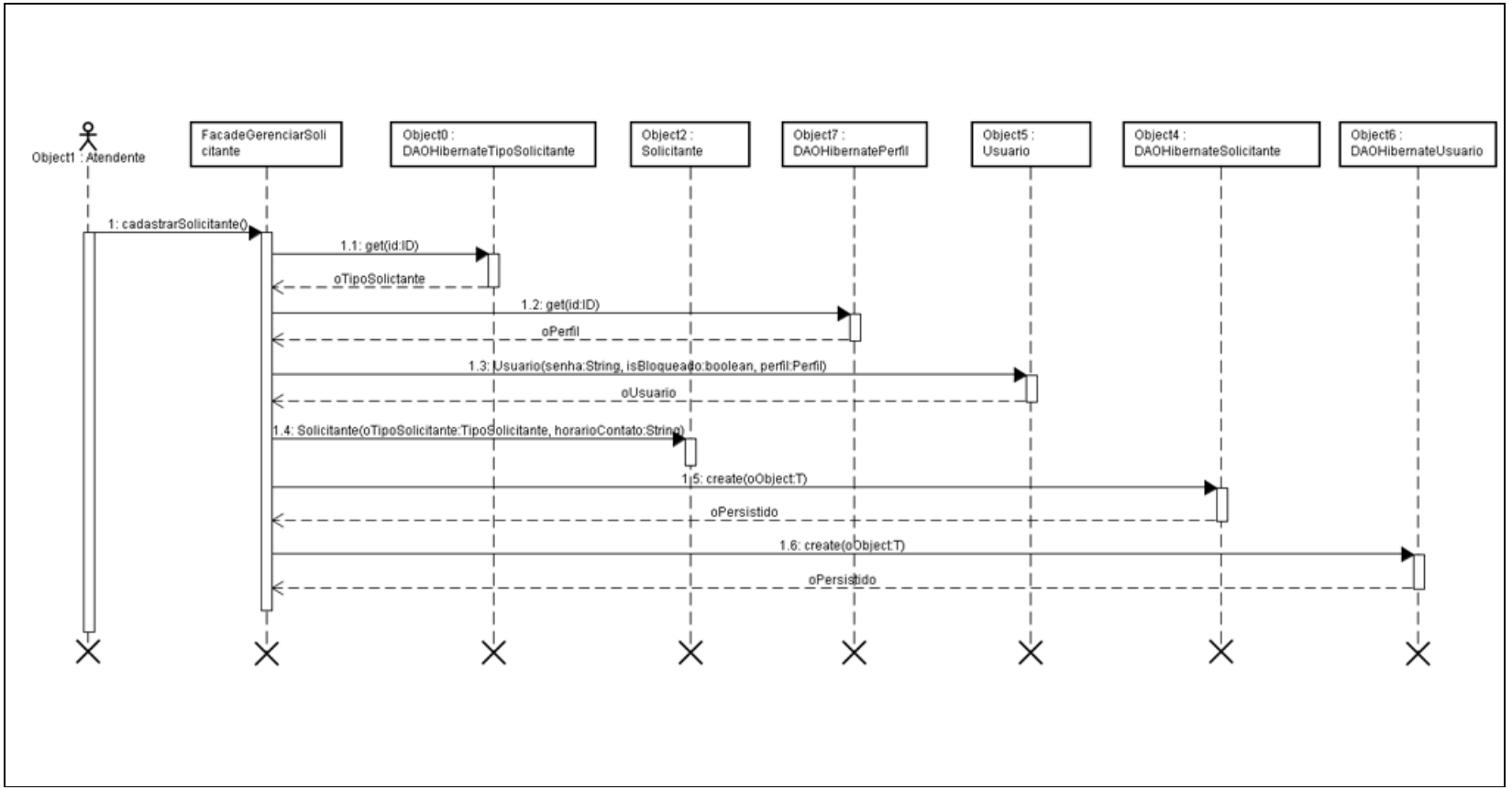
REABRIR OCORRÊNCIA



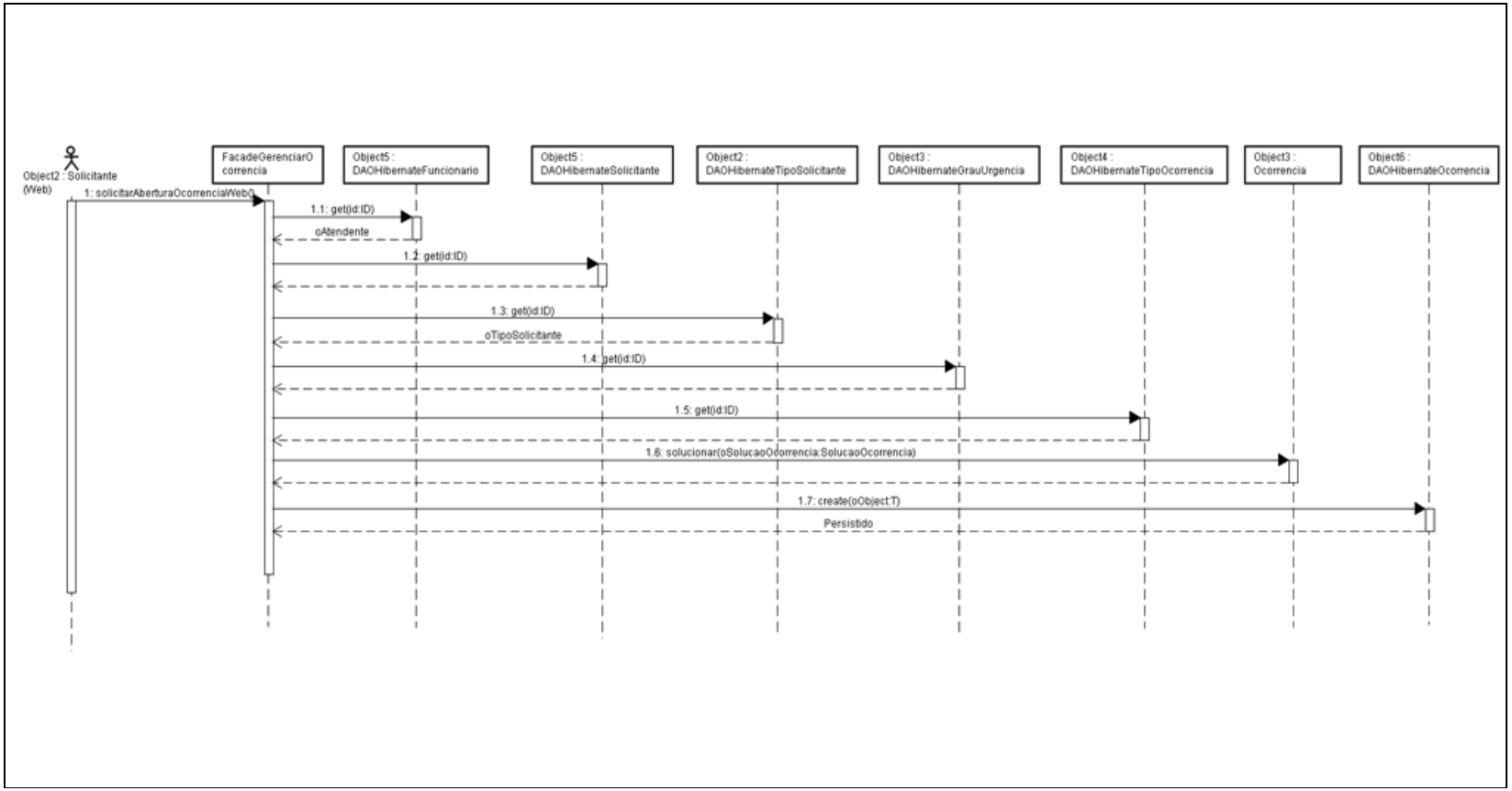
CADASTRAR CONTATO WEB



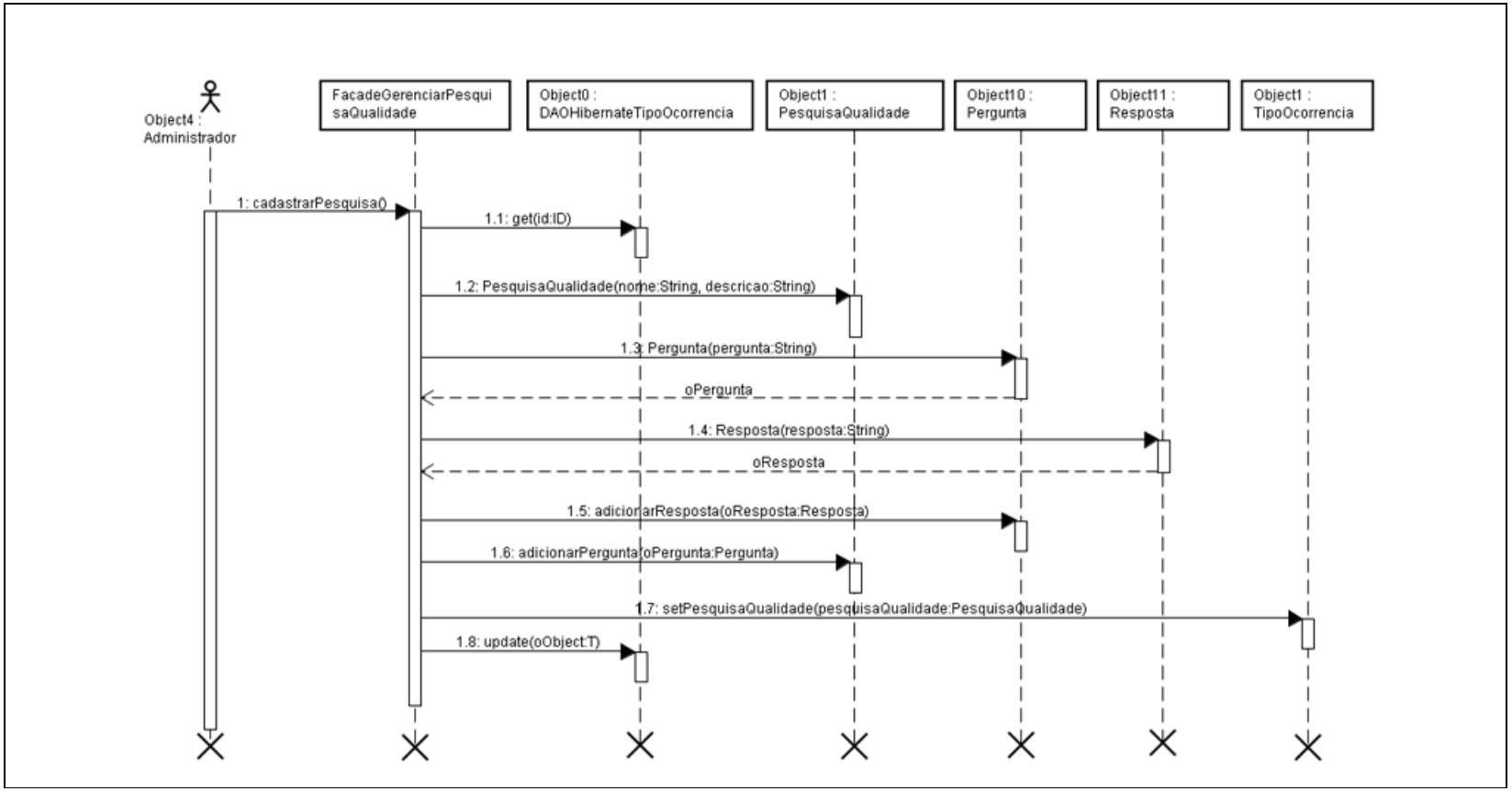
CADASTRAR SOLICITANTE WEB



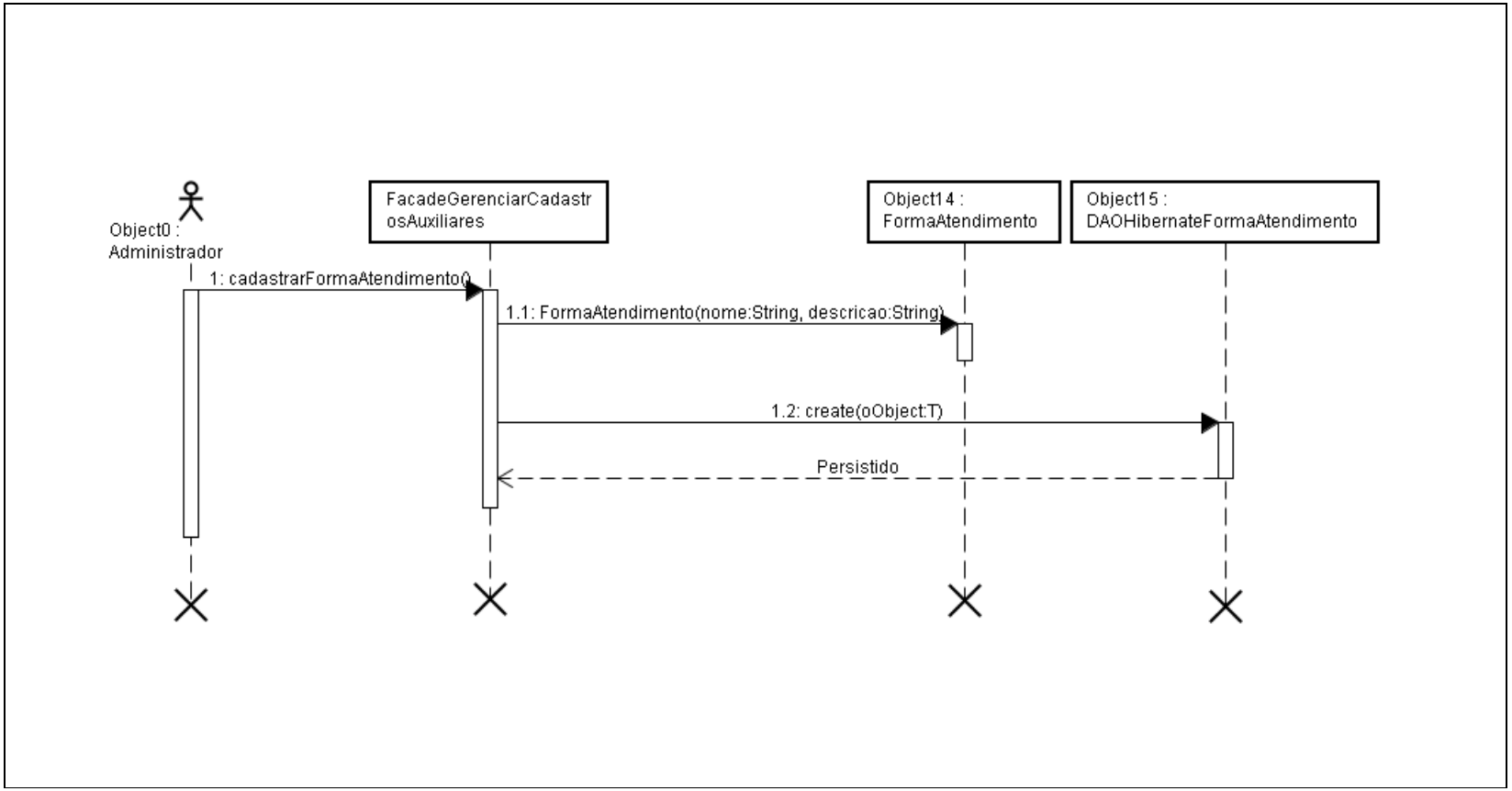
SOLICITAR ABERTURA DA OCORRÊNCIA WEB



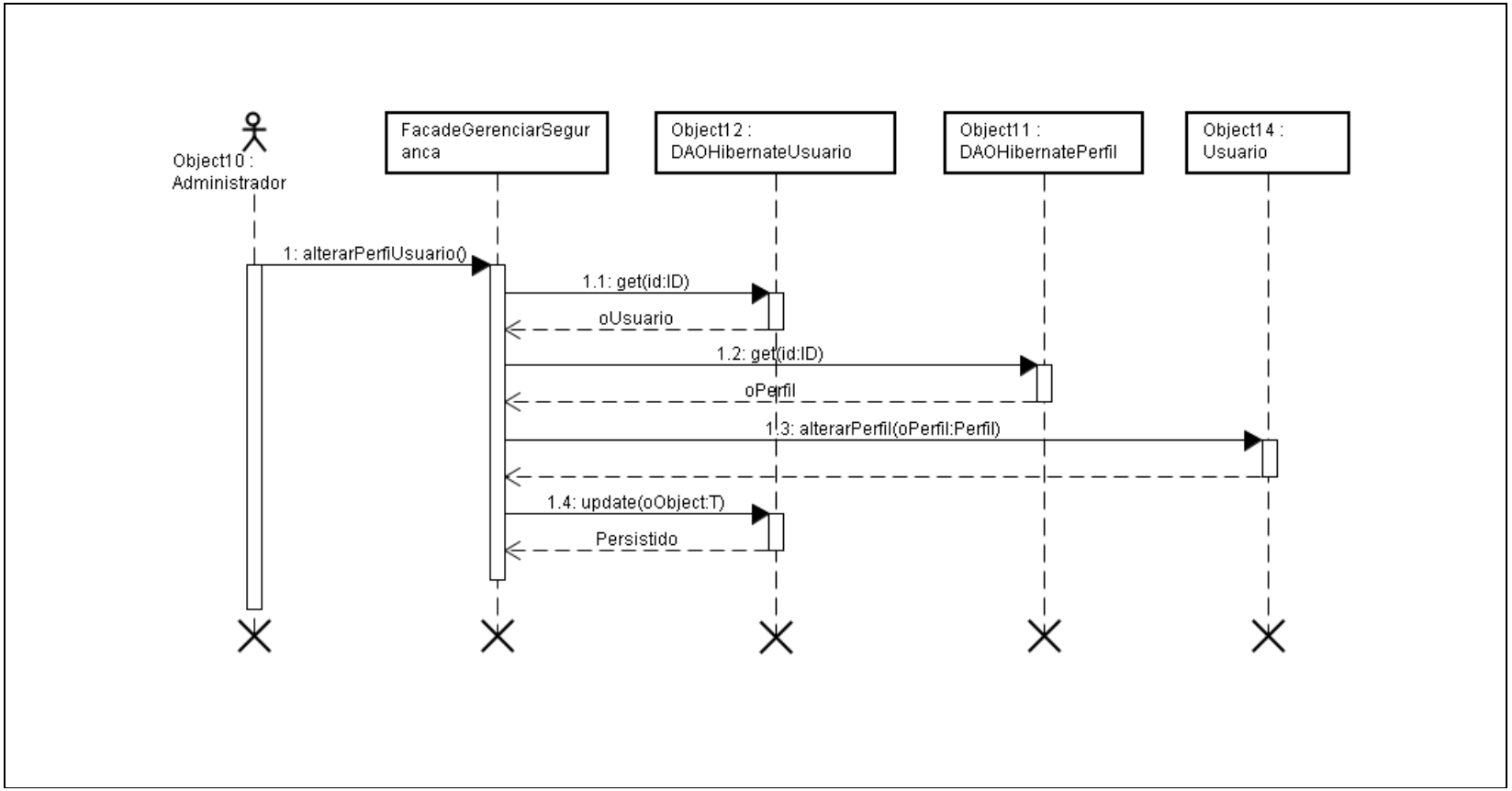
CADASTRAR PESQUISA DE QUALIDADE



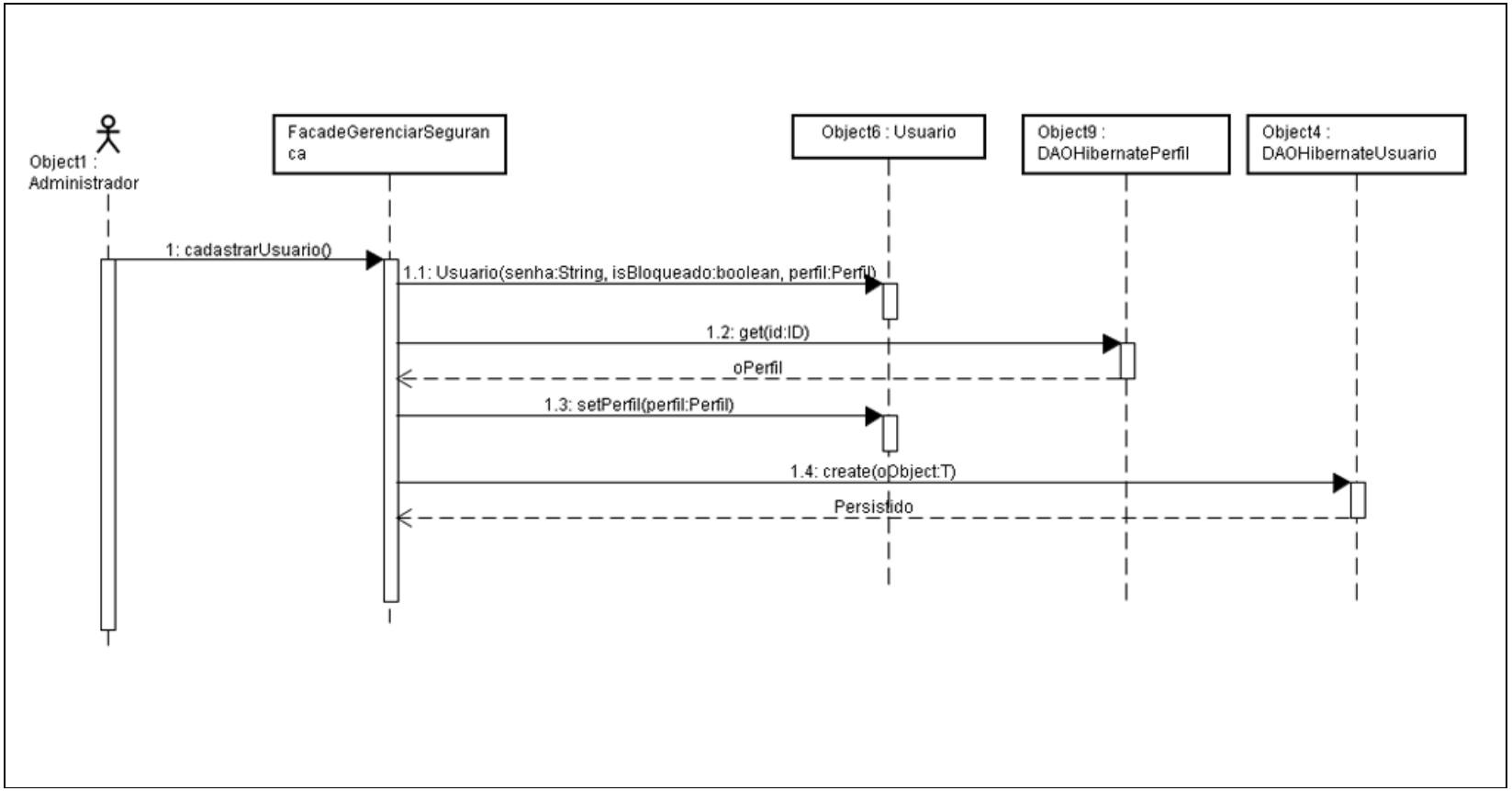
CADASTRAR FORMA DE ATENDIMENTO



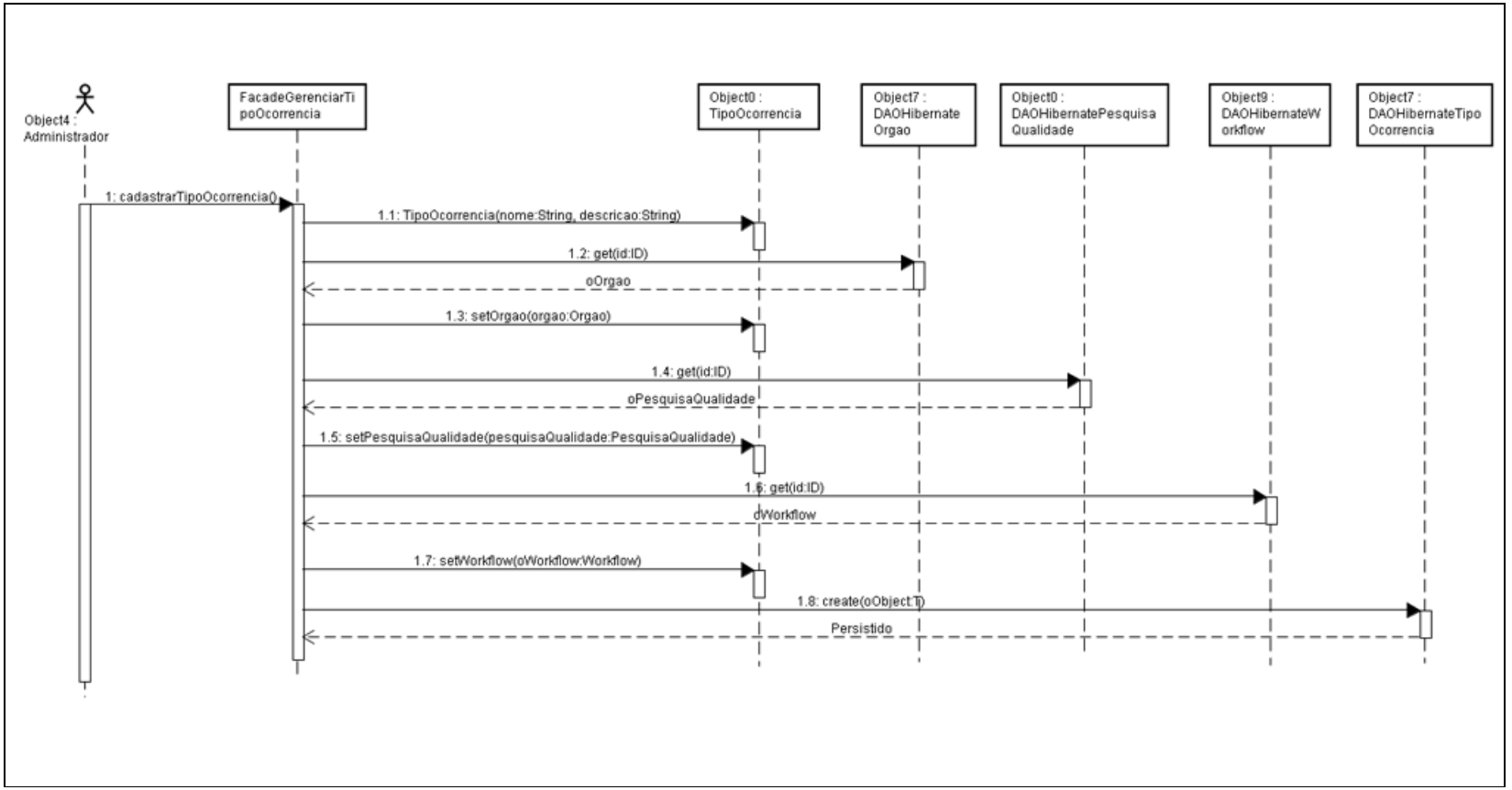
ALTERAR PERFIL DOS USUÁRIOS



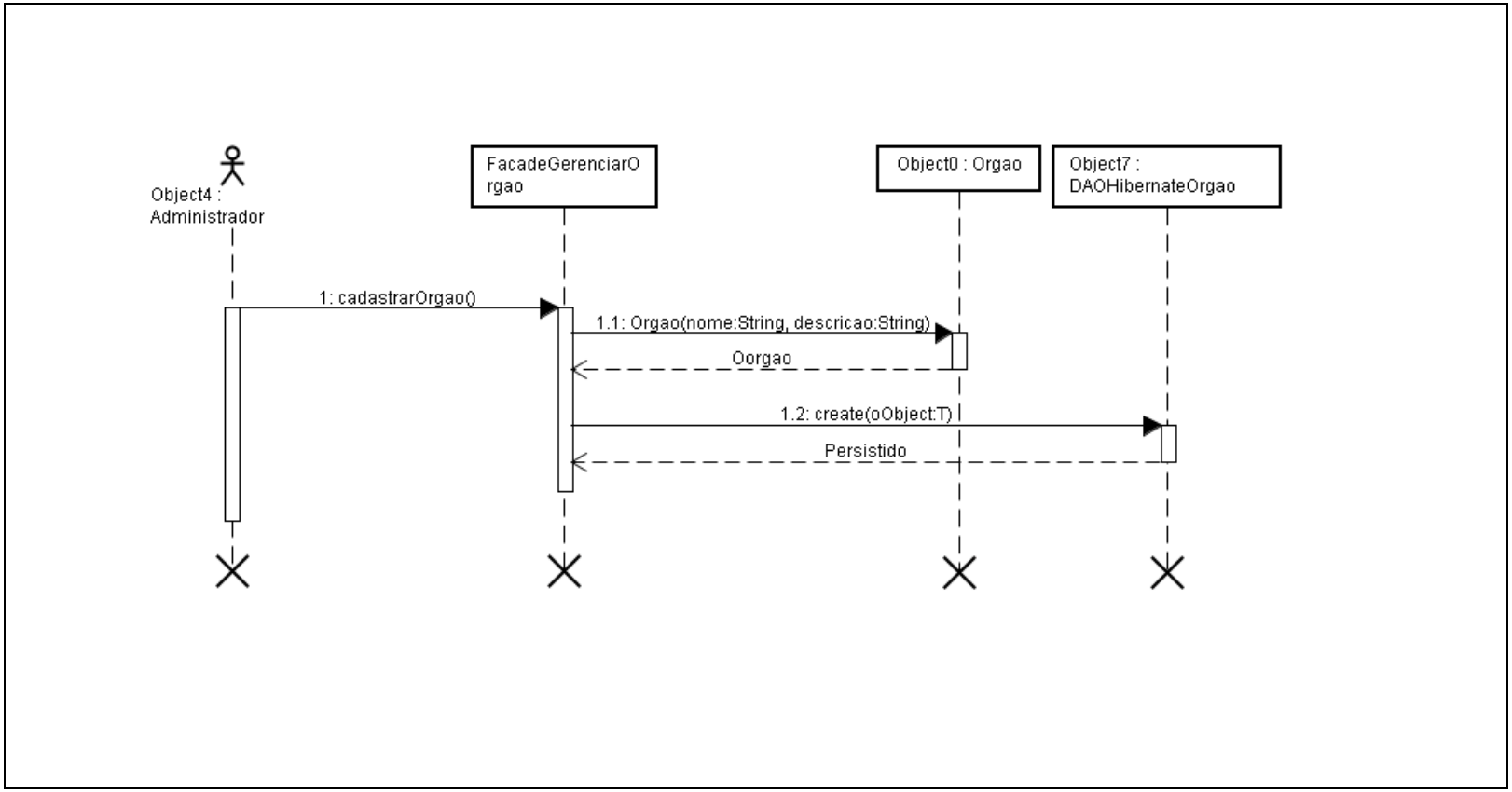
CADASTRAR USUÁRIOS



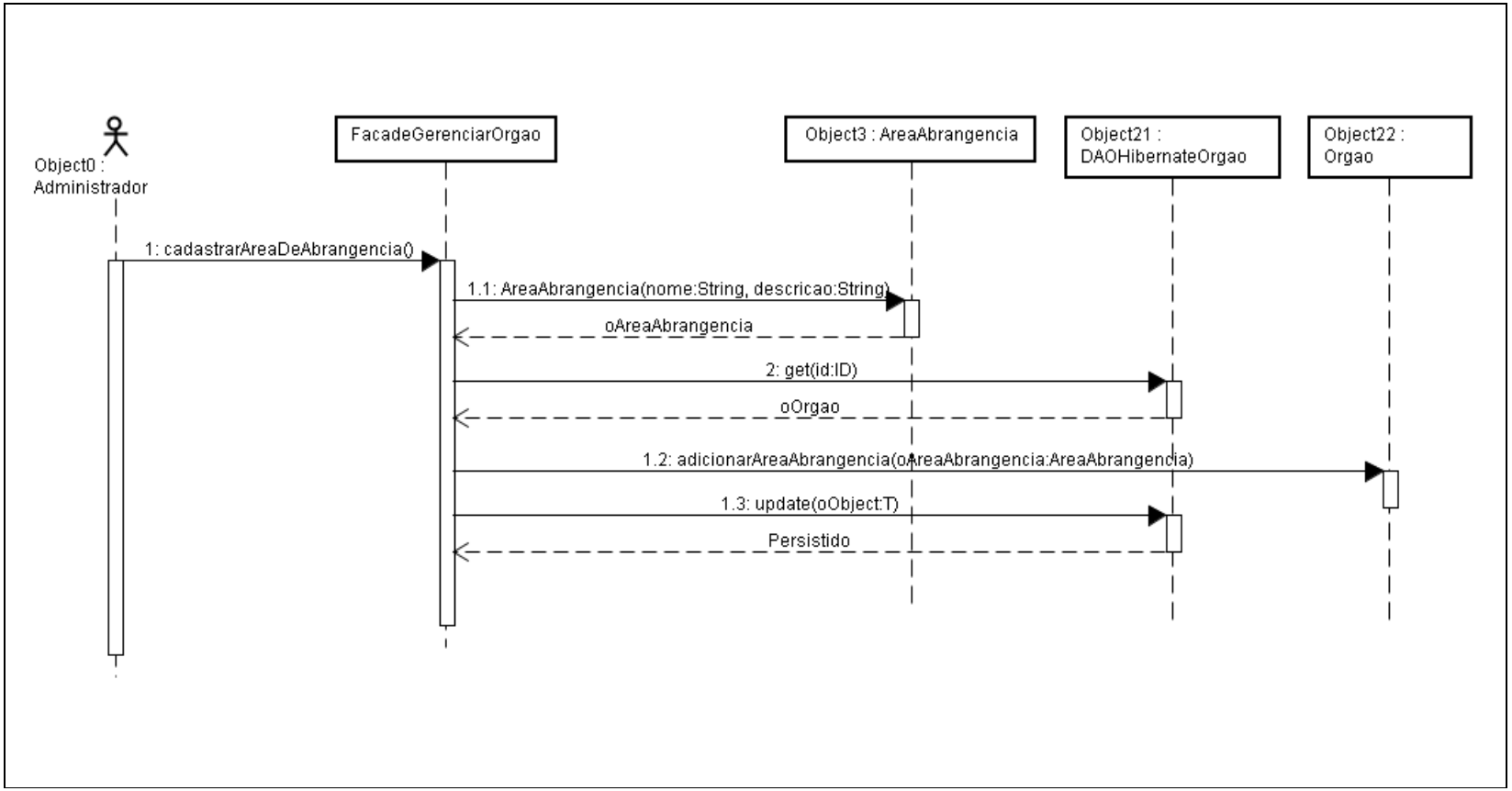
CADASTRAR TIPO OCORRÊNCIA



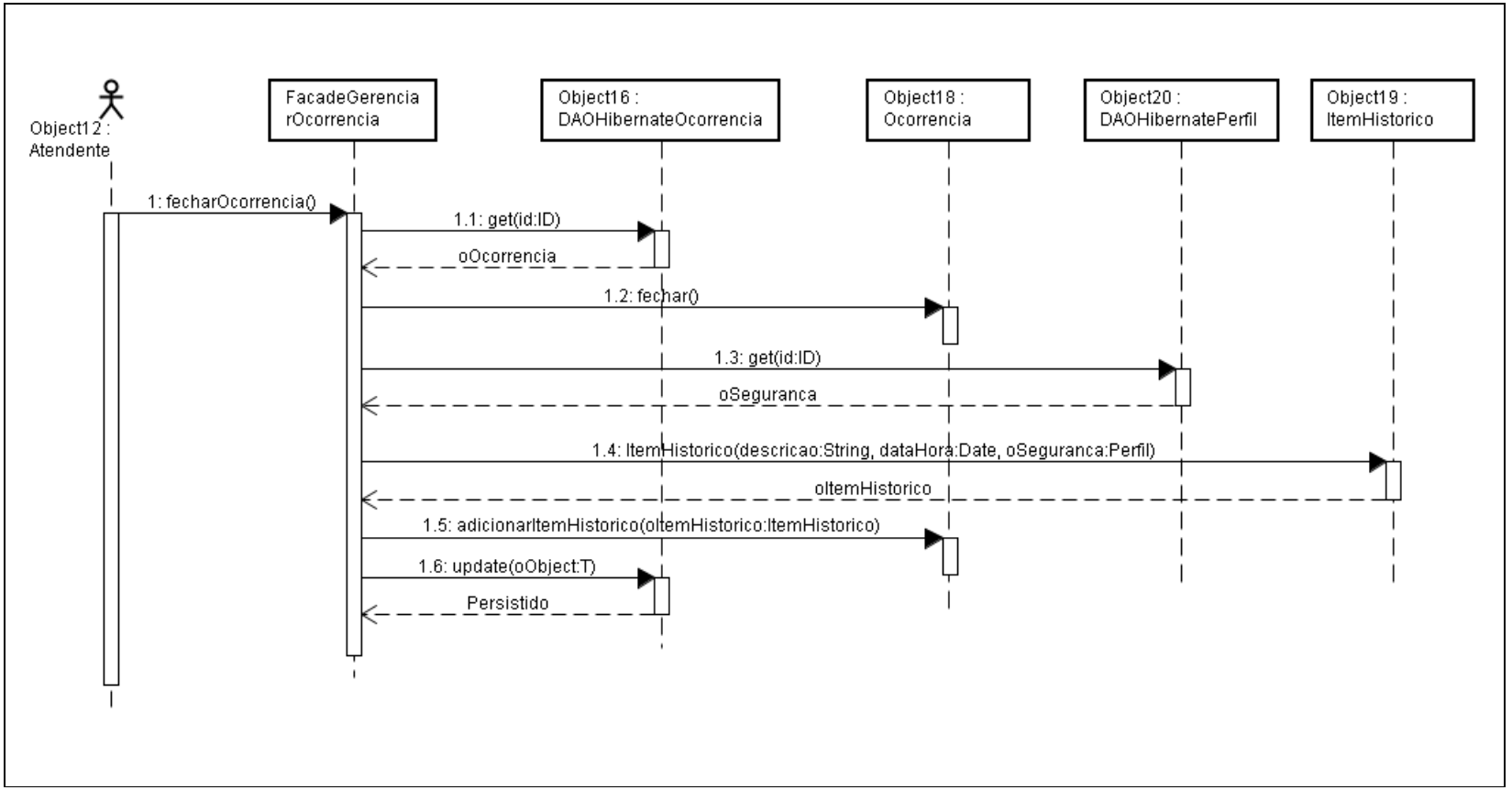
CADASTRAR ÓRGÃOS



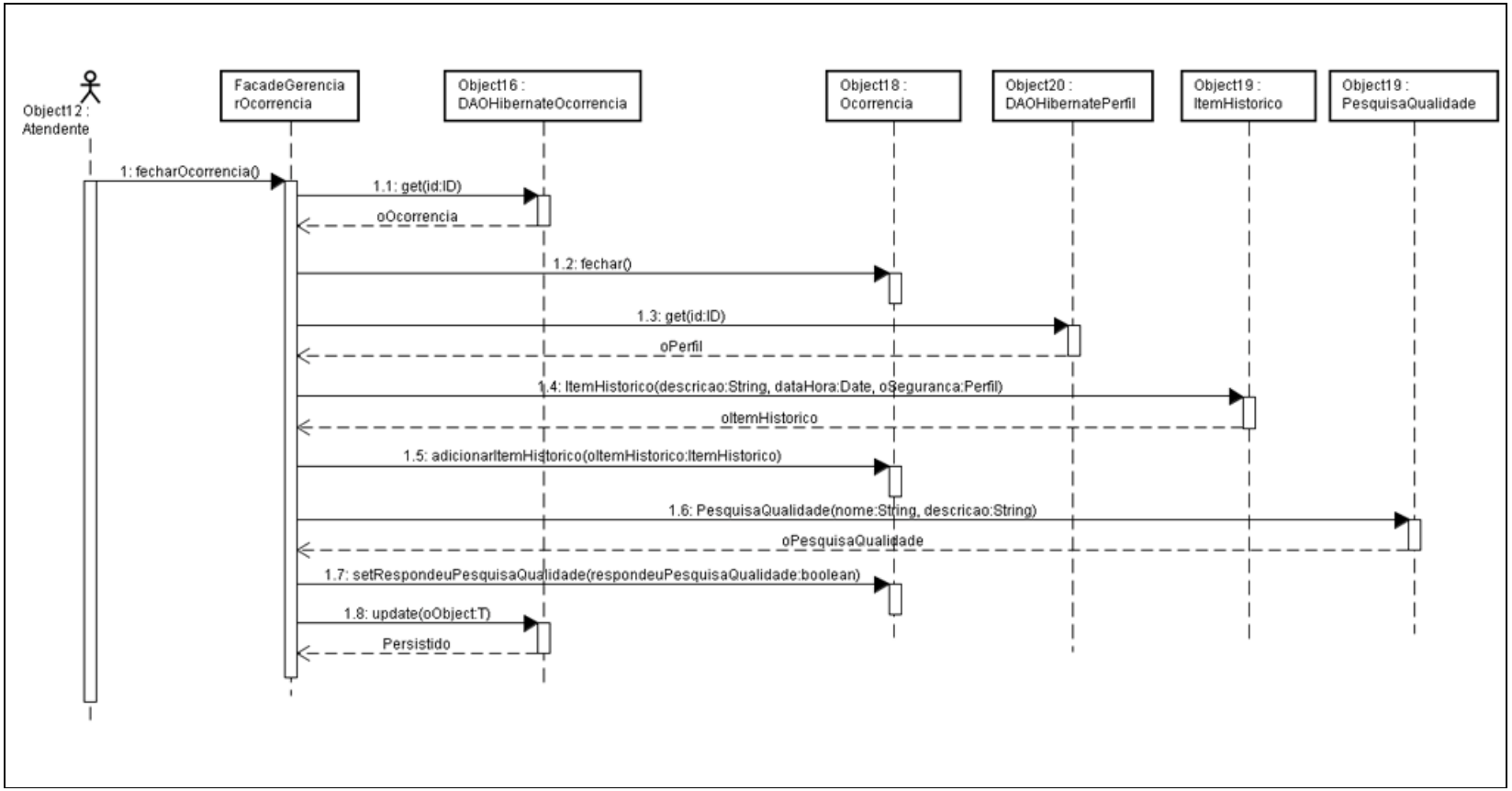
CADASTRAR ÁREAS DE ABRANGÊNCIA



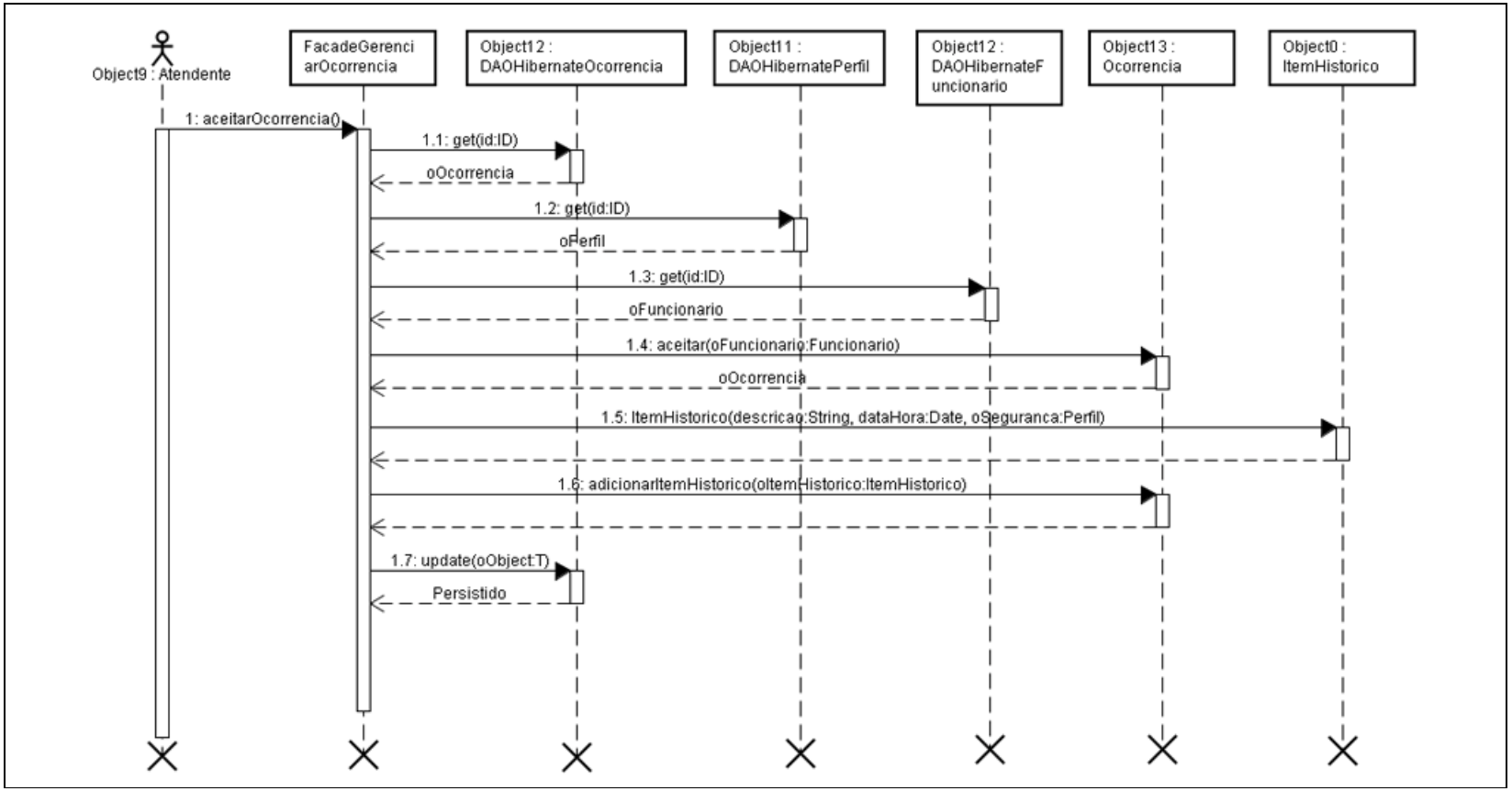
FECHAR OCORRÊNCIA



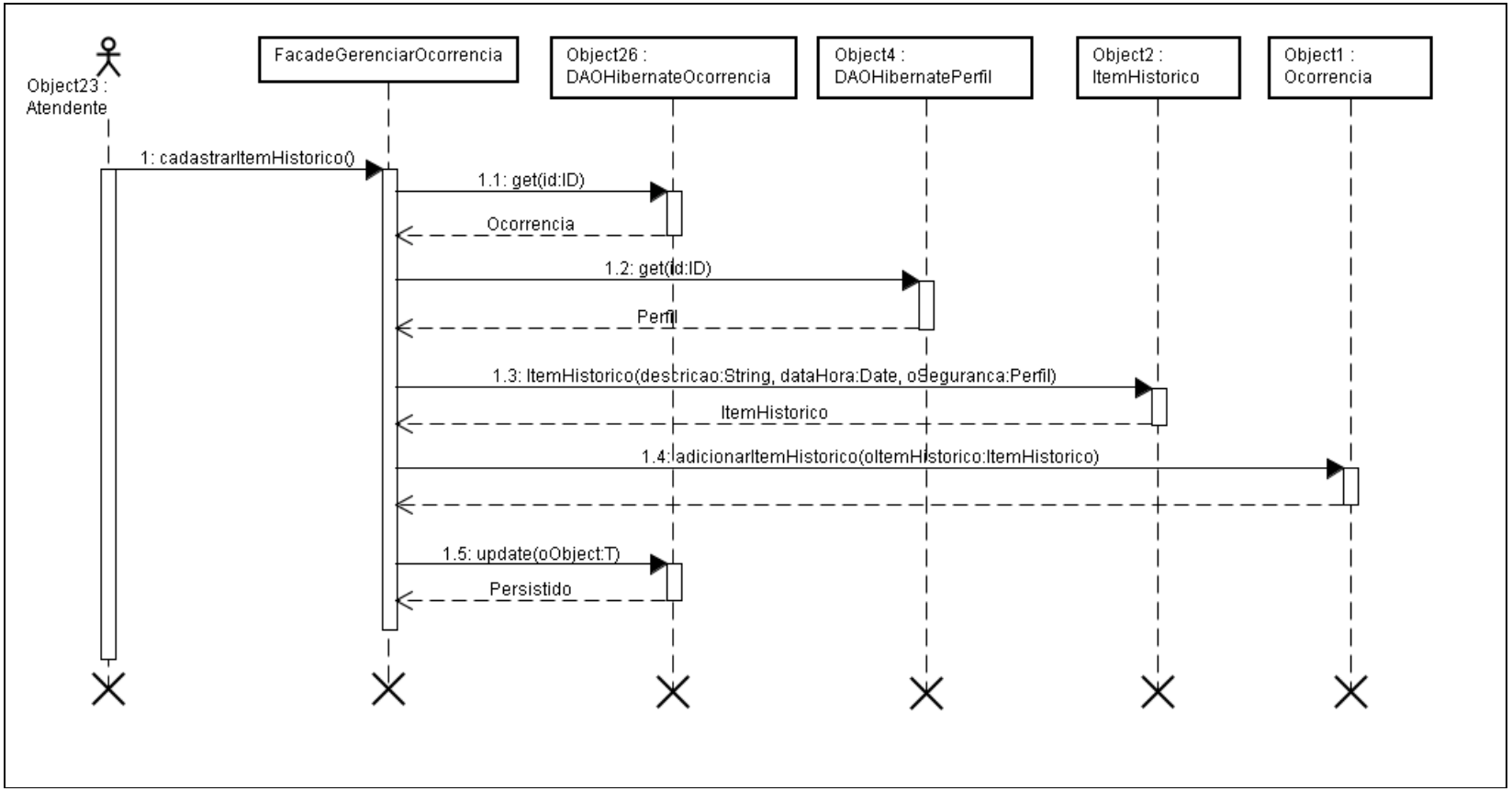
FECHAR OCORRÊNCIA - FLUXO ALTERNATIVO – REALIZAR PESQUISA DE QUALIDADE



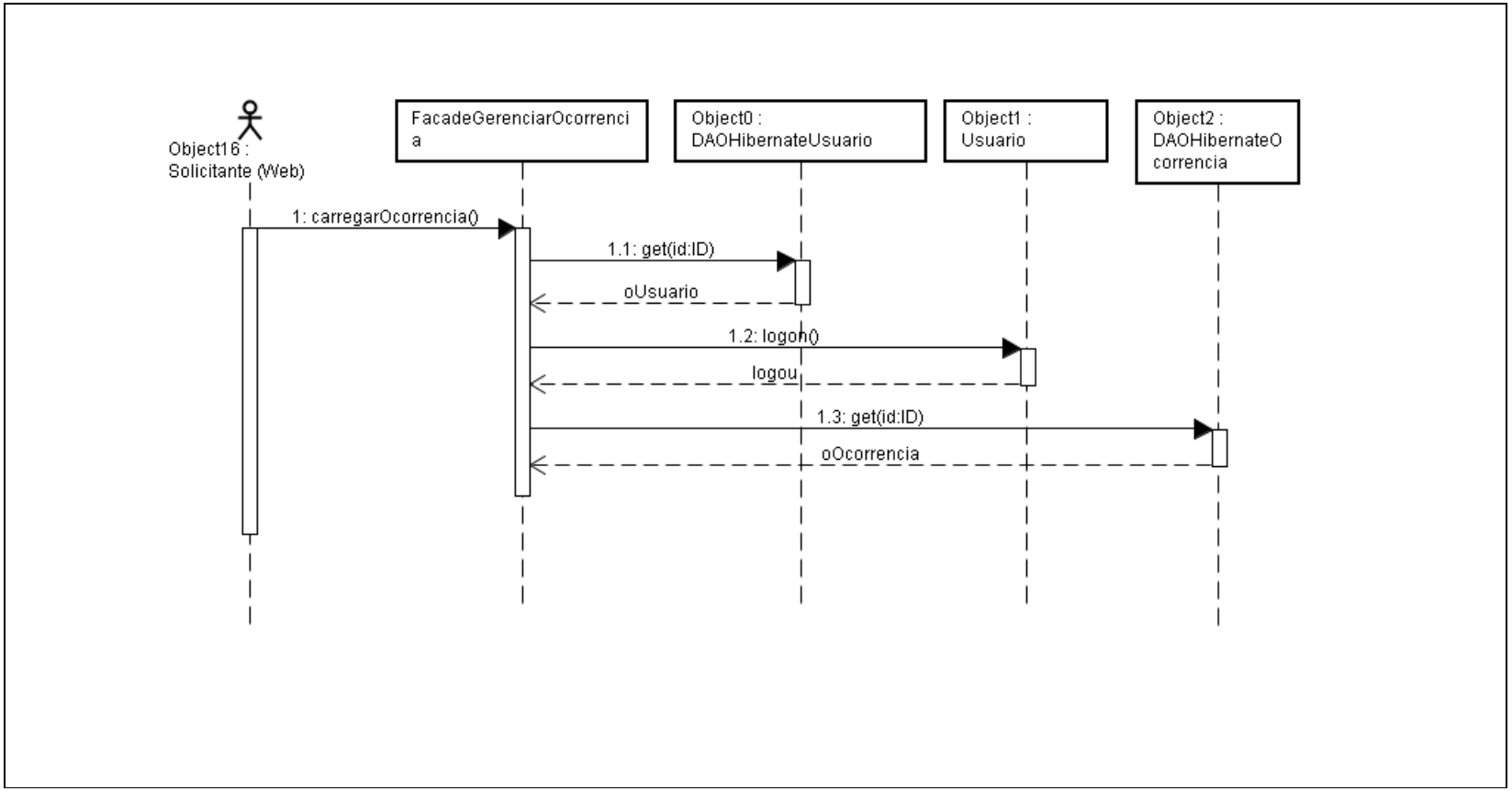
ACEITAR OCORRÊNCIA



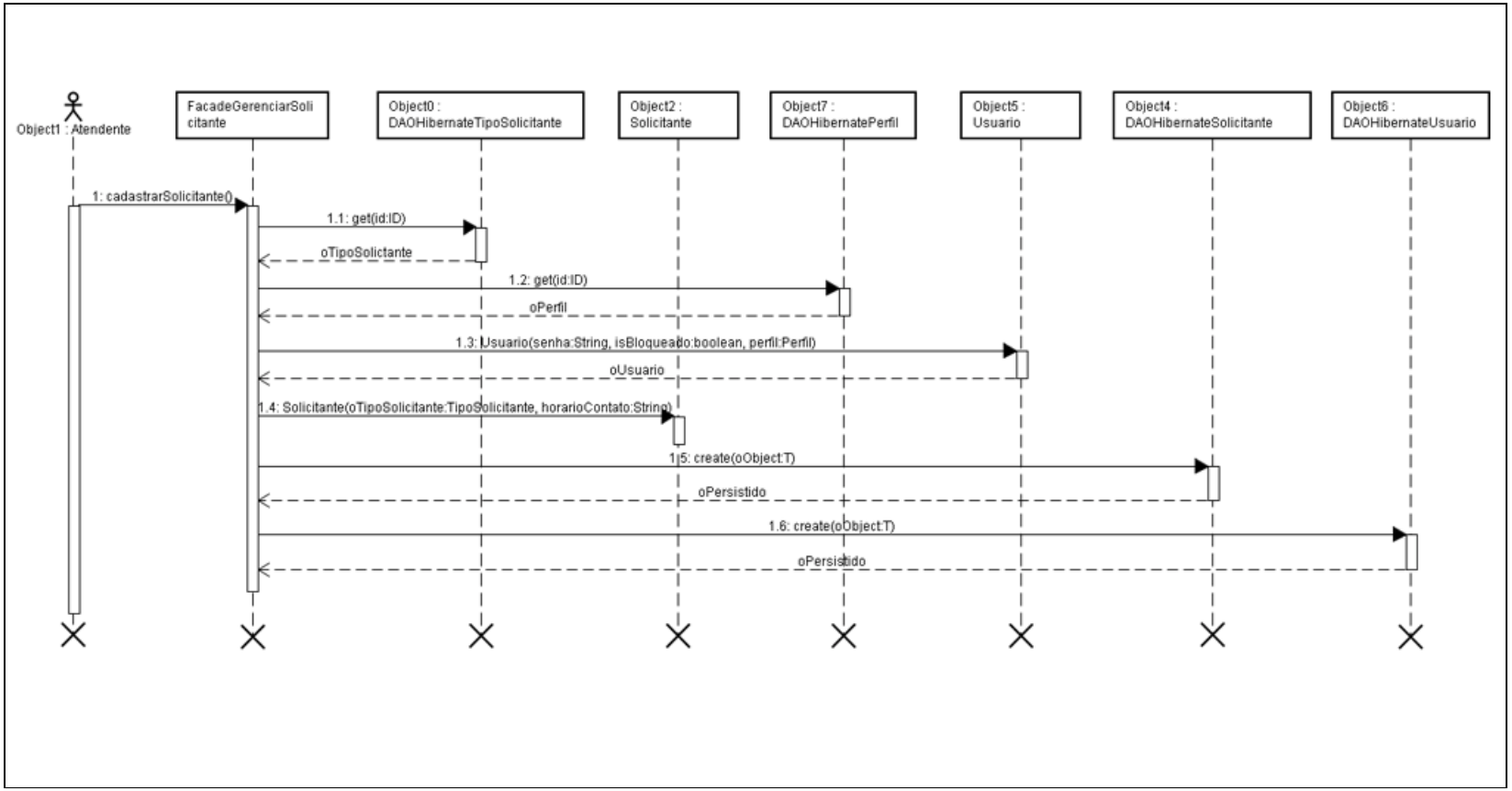
CADASTRAR ITEN DE HISTÓRICO DA OCORRÊNCIA



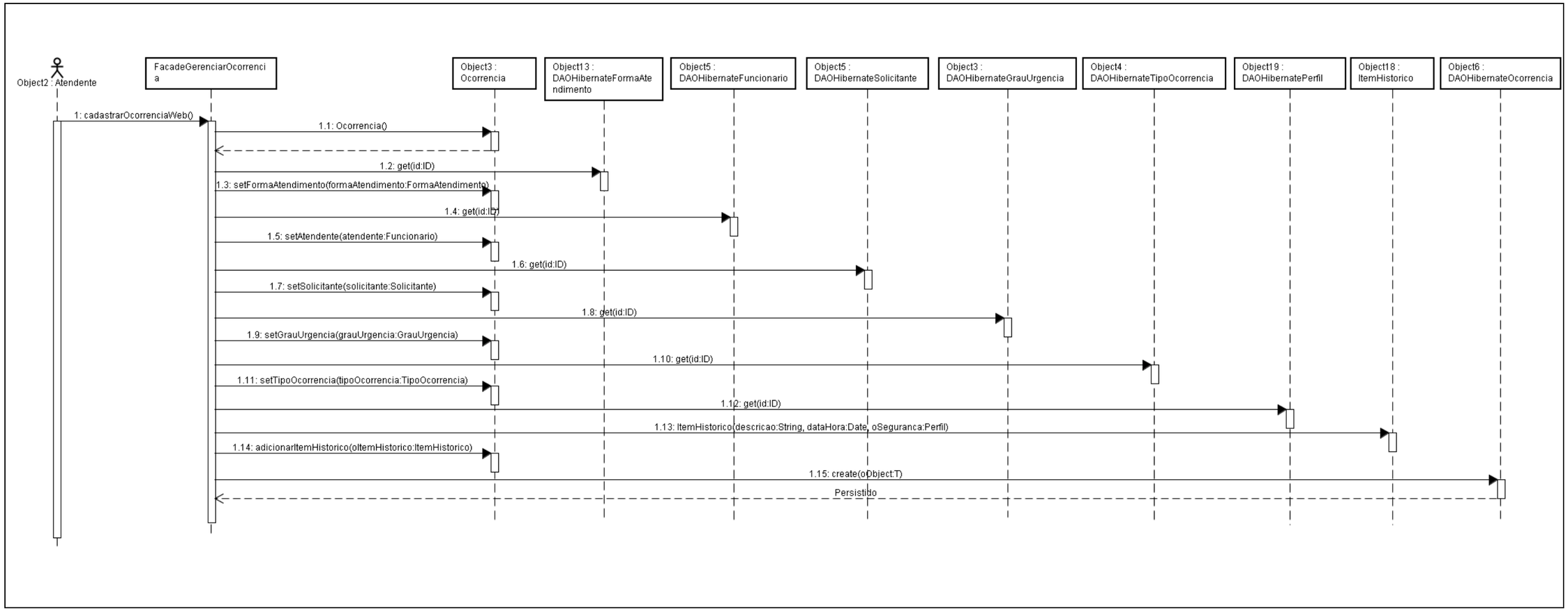
VISUALIZAR OCORRÊNCIA



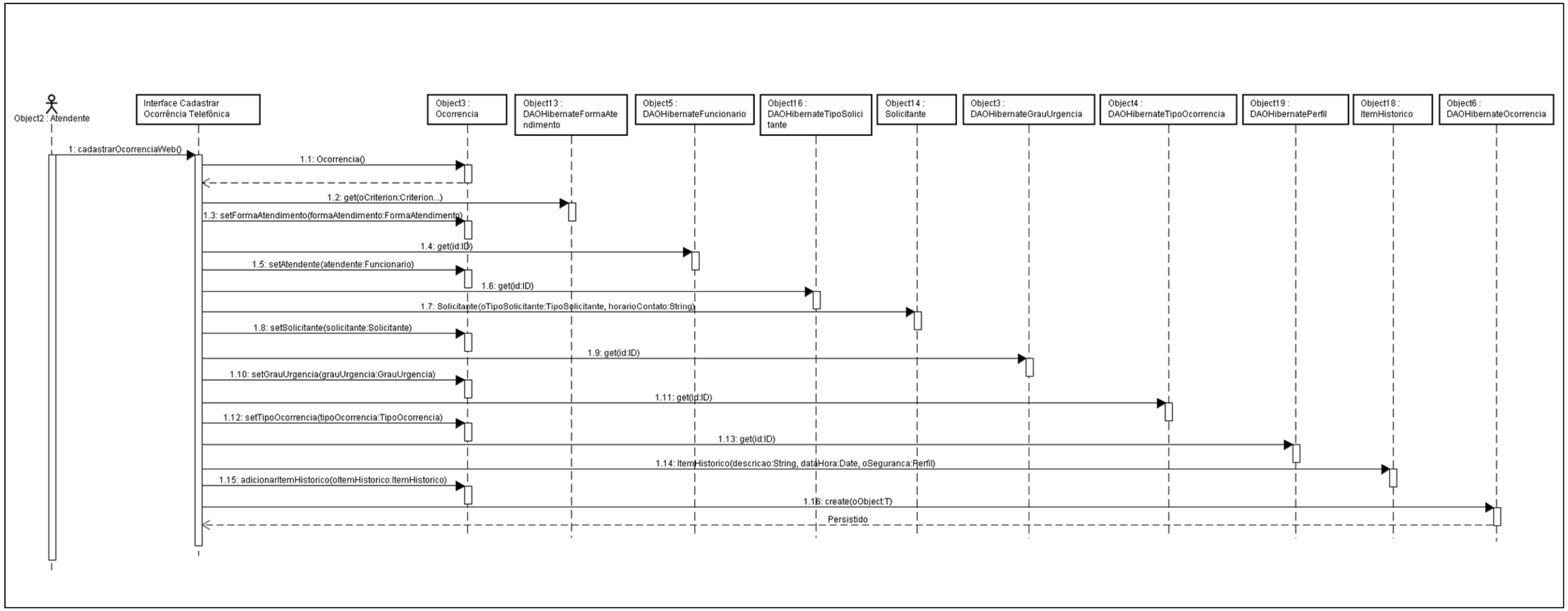
CADASTRAR SOLICITANTE



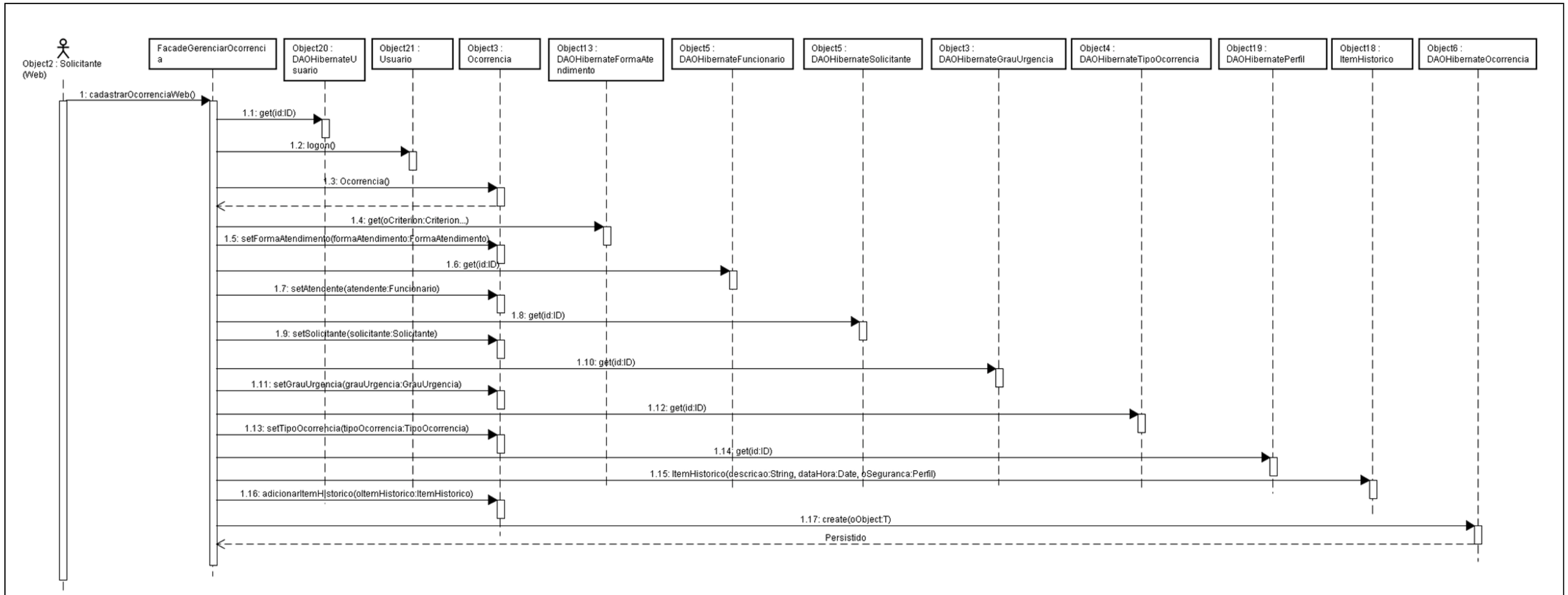
CADASTRAR OCORRÊNCIA TELEFÔNICA



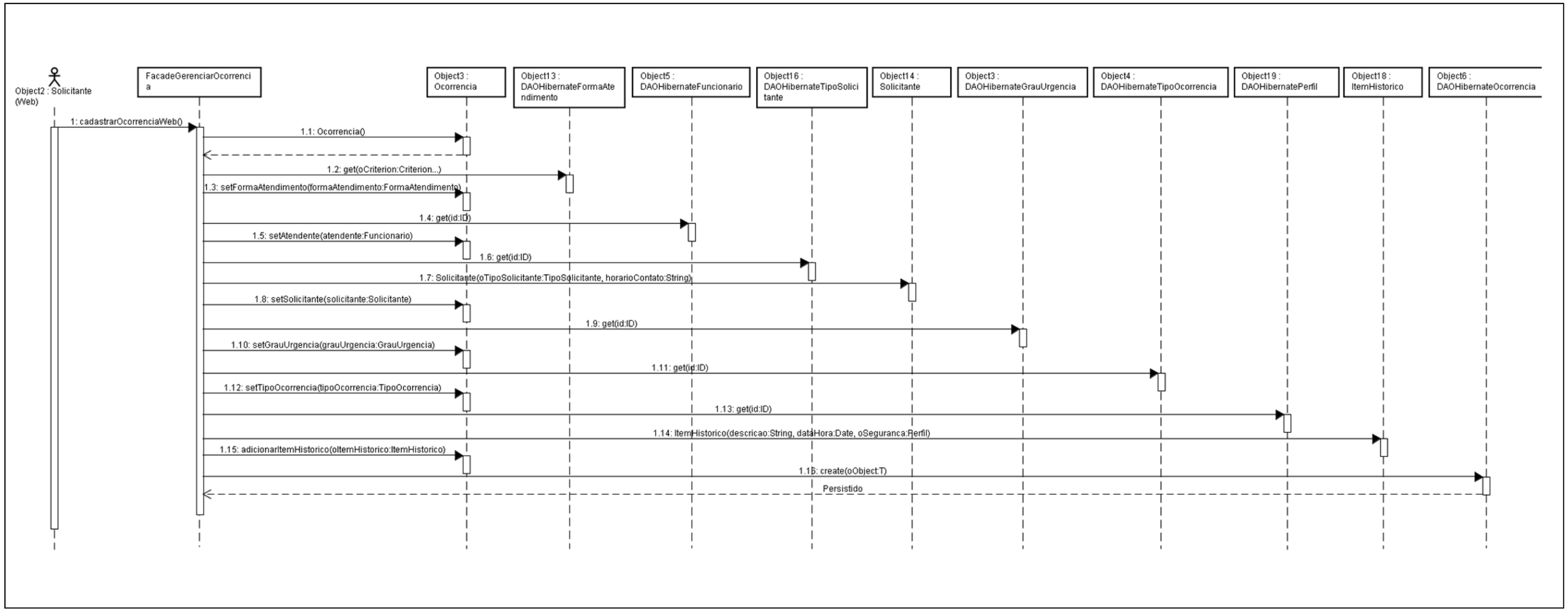
CADASTRAR OCORRÊNCIA TELEFÔNICA – FLUXO ALTERNATIVO SE O SOLICITANTE NÃO EXISTE



CADASTRAR OCORRÊNCIA WEB

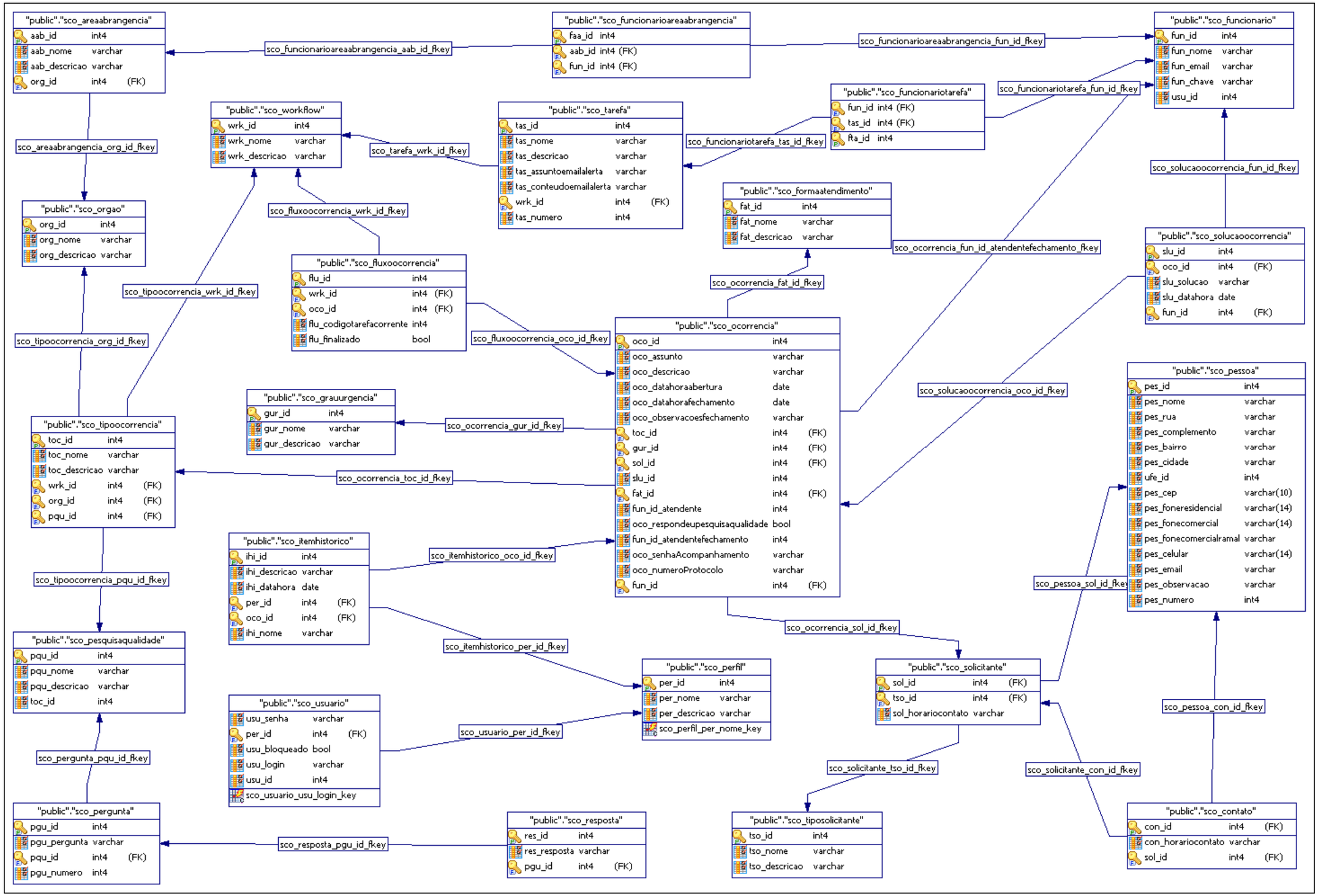


CADASTRAR OCORRÊNCIA WEB – FLUXO ALTERNATIVO SE O SOLICITANTE NÃO EXISTE



APÊNDICE III – MODELAGEM DE DADOS

DIAGRAMA ENTIDADE RELACIONAMENTO


"public"."sco_areaabrangencia"
aab_id int4
aab_nome varchar
aab_descricao varchar
org_id int4 (FK)
sco_funcionarioareaabrangencia_aab_id_fkey
"public"."sco_funcionarioareaabrangencia"
faa_id int4
aab_id int4 (FK)
fun_id int4 (FK)
sco_funcionarioareaabrangencia_fun_id_fkey
"public"."sco_funcionario"
fun_id int4
fun_nome varchar
fun_email varchar
fun_chave varchar
usu_id int4
sco_funcionariotarefa_fun_id_fkey
"public"."sco_funcionariotarefa"
fun_id int4 (FK)
tas_id int4 (FK)
fta_id int4
sco_funcionariotarefa_tas_id_fkey
"public"."sco_tarefa"
tas_id int4
tas_nome varchar
tas_descricao varchar
tas_assuntoemailalerta varchar
tas_conteudoemailalerta varchar
wrk_id int4 (FK)
tas_numero int4
"public"."sco_workflow"
wrk_id int4
wrk_nome varchar
wrk_descricao varchar
sco_tarefa_wrk_id_fkey
sco_areaabrangencia_org_id_fkey
"public"."sco_orgao"
org_id int4
org_nome varchar
org_descricao varchar
sco_fluxoocorrencia_wrk_id_fkey
"public"."sco_fluxoocorrencia"
flu_id int4
wrk_id int4 (FK)
oco_id int4 (FK)
flu_codigotarefacorrente int4
flu_finalizado bool
sco_tipoocorrencia_wrk_id_fkey
sco_tipoocorrencia_org_id_fkey
"public"."sco_tipoocorrencia"
toc_id int4
toc_nome varchar
toc_descricao varchar
wrk_id int4 (FK)
org_id int4 (FK)
pqu_id int4 (FK)
sco_tipoocorrencia_pqu_id_fkey
"public"."sco_grauurgencia"
gur_id int4
gur_nome varchar
gur_descricao varchar
sco_ocorrencia_gur_id_fkey
sco_ocorrencia_toc_id_fkey
sco_fluxoocorrencia_oco_id_fkey
"public"."sco_ocorrencia"
oco_id int4
oco_assunto varchar
oco_descricao varchar
oco_datahoraabertura date
oco_datahorafechamento date
oco_observacoesfechamento varchar
toc_id int4 (FK)
gur_id int4 (FK)
sol_id int4 (FK)
slu_id int4
fat_id int4 (FK)
fun_id_atendente int4
oco_respondeupesquisaqualidade bool
fun_id_atendentefechamento int4
oco_senhaAcompanhamento varchar
oco_numeroProtocolo varchar
fun_id int4 (FK)
"public"."sco_formaatendimento"
fat_id int4
fat_nome varchar
fat_descricao varchar
sco_ocorrencia_fat_id_fkey
sco_ocorrencia_fun_id_atendentefechamento_fkey
sco_solucaoocorrencia_fun_id_fkey
"public"."sco_solucaoocorrencia"
slu_id int4
oco_id int4 (FK)
slu_solucao varchar
slu_datahora date
fun_id int4 (FK)
sco_solucaoocorrencia_oco_id_fkey
"public"."sco_pessoa"
pes_id int4
pes_nome varchar
pes_rua varchar
pes_complemento varchar
pes_bairro varchar
pes_cidade varchar
ufe_id int4
pes_cep varchar(10)
pes_foneresidencial varchar(14)
pes_fonecomercial varchar(14)
pes_fonecomercialramal varchar
pes_celular varchar(14)
pes_email varchar
pes_observacao varchar
pes_numero int4
sco_pessoa_sol_id_fkey
"public"."sco_itemhistorico"
ihi_id int4
ihi_descricao varchar
ihi_datahora date
per_id int4 (FK)
oco_id int4 (FK)
ihi_nome varchar
sco_itemhistorico_oco_id_fkey
sco_itemhistorico_per_id_fkey
sco_ocorrencia_sol_id_fkey
"public"."sco_pesquisaqualidade"
pqu_id int4
pqu_nome varchar
pqu_descricao varchar
toc_id int4
"public"."sco_perfil"
per_id int4
per_nome varchar
per_descricao varchar
sco_perfil_per_nome_key
"public"."sco_solicitante"
sol_id int4 (FK)
tso_id int4 (FK)
sol_horariocontato varchar
sco_pessoa_con_id_fkey
"public"."sco_usuario"
usu_senha varchar
per_id int4 (FK)
usu_bloqueado bool
usu_login varchar
usu_id int4
sco_usuario_usu_login_key
sco_usuario_per_id_fkey
sco_solicitante_tso_id_fkey
sco_solicitante_con_id_fkey
sco_pergunta_pqu_id_fkey
"public"."sco_pergunta"
pgu_id int4
pgu_pergunta varchar
pqu_id int4 (FK)
pgu_numero int4
"public"."sco_resposta"
res_id int4
res_resposta varchar
pgu_id int4 (FK)
sco_resposta_pgu_id_fkey
"public"."sco_tiposolicitante"
tso_id int4
tso_nome varchar
tso_descricao varchar
"public"."sco_contato"
con_id int4 (FK)
con_horariocontato varchar
sol_id int4 (FK)

DICIONÁRIO DE DADOS

Tabela sco_tiposolicitante				
Descrição: Tabela contendo dados referentes ao solicitante				
Campos sco_tiposolicitante				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
tso_id	SERIAL	Numero seqüencial para identificar o tipo de solicitante	x	
tso_nome	Varchar(255)	Nome do tipo de solicitante		
tso_descricao	Varchar(255)	Descrição do tipo de solicitante		
Permissões sco_tiposolicitante				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_usuario				
Descrição: Tabela contendo dados referentes ao usuário.				
Campos sco_usuario				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
usu_id	SERIAL	Número seqüencial para identificar o usuário	x	
per_id	INTEGER	Código do perfil		x (com a coluna per_id em sco_perfil)
usu_login	Varchar(20)	Login do usuário		
usu_senha	Varchar(20)	Senha do usuário		
usu_bloqueado	Boolean	Usuário bloqueado ou não		
Permissões sco_usuario				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_workflow				
Descrição: Tabela contendo dados referentes ao workflow				
Campos sco_workflow				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
wrk_id	SERIAL	Número seqüencial para identificar o workflow	x	
wrk_nome	Varchar(255)	Nome do workflow		
wrk_descricao	Varchar(255)	Descrição do workflow		
Permissões sco_workflow				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_tipoocorrencia				
Descrição: Tabela contendo dados referentes ao tipo da ocorrência				
Campos sco_tipoocorrencia				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
toc_id	SERIAL	Número seqüencial para identificar o tipo da ocorrência	x	
toc_nome	Varchar(255)	Nome do tipo da ocorrência		
toc_descricao	Varchar(255)	Descrição do tipo da ocorrência		
wrk_id	INTEGER	Código do workflow		x (com a coluna wrk_id em sco_workflow)
org_id	INTEGER	Código do Órgão		
pqu_id	INTEGER	Código da Pesquisa de Qualidade		
Permissões sco_tipoocorrencia				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_tarefas				
Descrição: Tabela contendo dados referentes as tarefas.				
Campos sco_tarefas				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
tas_id	SERIAL	Número seqüencial para identificar a tarefa	x	
wrk_id	INTEGER	Código do workflow		x (com a coluna wrk_id em sco_workflow)
tas_nome	Varchar(255)	Nome da tarefa		
tas_descricao	Varchar(255)	Descrição da tarefa		
tas_assuntoemailalerta	Varchar(255)	Assunto do e-mail de alerta		
tas_conteudoemailalerta	TEXT	Conteúdo do e-mail de alerta		
tas_numero	INTEGER	Número de ordem da tarefa		
Permissões sco_tarefas				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_orgao				
Descrição: Tabela contendo dados referentes ao orgão				
Campos sco_orgao				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
org_id	SERIAL	Número seqüencial para identificar o orgão	x	
org_nome	Varchar(40)	Nome do orgão		
org_descricao	Varchar(255)	Descrição do orgão		
Permissões sco_orgao				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_funcionariotarefa				
Descrição: Tabela que relaciona os dados das tabelas sco_tarefas e sco_funcionario				
Campos sco_funcionariotarefa				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
fta_id	SERIAL	Número seqüencial para identificar a relação funcionário e tarefa	x	
tas_id	INTEGER	Código tarefa		x (com tas_id em sco_tarefa)
fun_id	INTEGER	Código do funcionário		x (com fun_id em sco_funcionario)
Permissões sco_funcionariotarefa				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_itemhistorico				
Descrição: Tabela contendo dados referentes ao histórico.				
Campos sco_itemhistorico				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
ihi_id	SERIAL	Número seqüencial para identificar o item do histórico	x	
ihi_nome	Varchar(255)	Nome do item de histórico		
ihi_descricao	Varchar(255)	Descrição do item de histórico		
ihi_datahora	DATE	Data e hora do item de histórico		
per_id	INTEGER	Código do perfil		x (com a coluna per_id em sco_perfil)
oco_id	INTEGER	Código da ocorrência		x (com a coluna oco_id em sco_ocorrencia)
Permissões sco_itemhistorico				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_funcionario				
Descrição: Tabela contendo dados referentes ao funcionário.				
Campos sco_funcionario				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
fun_id	SERIAL	Número seqüencial para identificar o funcionário	x	
fun_chave	Varchar(255)	Chave do funcionário		
fun_nome	Varchar(255)	Nome do funcionário		
fun_email	Varchar(255)	E-mail do funcionário		
usu_id	INTEGER	Código do Usuário		x (com a coluna usu_id em sco_usuario
Permissões sco_funcionario				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_areaabrangencia				
Descrição: Tabela contendo dados referentes à área de abrangência.				
Campos sco_areaabrangencia				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
aab_id	SERIAL	Número seqüencial para identificar a área de abrangência	x	
aab_nome	Varchar(255)	Nome da área de abrangência		
aab_descricao	Varchar(255)	Descrição da área de abrangência		
org_id	INTEGER	Código do órgão		x (com a coluna org_id em sco_orgao)
Permissões sco_areaabrangencia				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_solicitante				
Descrição: Tabela contendo dados referentes ao solicitante.				
Campos sco_solicitante				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
sol_id	SERIAL	Número seqüencial para identificar o solicitante	x	
tso_id	INTEGER	Código do tipo de solicitante		x (com a coluna tso_id em sco_tiposolicitante)
sol_horariocontato	Varchar(255)	Horário de contato do solicitante		
Permissões sco_solicitante				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_pessoa				
Descrição: Tabela contendo dados referentes à pessoa.				
Campos sco_pessoa				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
pes_id	SERIAL	Número seqüencial para identificar a pessoa	x	
pes_nome	Varchar(255)	Nome da pessoa		
pes_rua	Varchar(255)	Rua da pessoa		
pes_numero	INTEGER	Numero na rua da pessoa		
pes_complemento	Varchar(255)	Complemento da rua da pessoa		
pes_bairro	Varchar(255))	Bairro da pessoa		
pes_cidade	Varchar(255)	Cidade da pessoa		
pes_estado	Varchar(255)	Estado da pessoa		
pes_cep	Varchar(9)	Cep da pessoa		
pes_foneresidencial	Varchar(13)	Telefone residencial da pessoa		
pes_fonecomercial	Varchar(13)	Telefone comercial da pessoa		
pes_fonecomercialramal	Varchar(4)	Ramal do telefone comercial da pessoa		
pes_celular	Varchar(13)	Telefone celular da pessoa		
pes_email	Varchar(255)	E-mail da pessoa		
pes_observacao	Varchar(255)	Observação referente à pessoa		
Permissões sco_pessoa				
USUÁRIO		PERMISSÕES DE BANCO		
sco		arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)		

Tabela sco_ocorrencia				
Descrição: Tabela contendo dados referentes à ocorrência.				
Campos sco_ocorrencia				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
oco_id	SERIAL	Número seqüencial para identificar a ocorrência	x	
toc_id	INTEGER	Código do tipo de ocorrência		x (com coluna toc_id em sco_tipoocorrencia)
gur_id	INTEGER	Código do grau de urgência		x (com a coluna gur_id em sco_grauurgencia)
slu_id	INTEGER	Código da solução da ocorrência		x (com a coluna slu_id em sco_solucaoocorrencia)
fat_id	INTEGER	Código da forma de atendimento		x (com a coluna fat_id em sco_formaatendimento)
fun_id_atendente	INTEGER	Código do funcionário atendente		x (com a coluna fun_id em sco_funcionario)
fun_id_atendente fechamento	INTEGER	Código do funcionário atendente de fechamento		x (com a coluna fun_id em sco_funcionario)
oco_datahora abertura	DATE	Data e hora de abertura da ocorrência		
oco_assunto	Varchar(255)	Assunto da ocorrência		
oco_descricao	Varchar(255)	Descrição da ocorrência		
oco_datasolucao	DATE	Data da solução da ocorrência		
oco_respondeu pesquisaqualidade	Boolean	Se a pesquisa de qualidade foi respondida		
oco_datahora fechamento	DATE	Data e hora do fechamento da ocorrência		
oco_observacao	Varchar(255)	Observação no		

fechamento		fechamento da ocorrência		
oco_protocolo	Varchar(255)	Protocolo da ocorrência		
oco_senha	Varchar(20)	Senha para acompanhamento da ocorrência		
Permissões sco_ocorrencia				
USUÁRIO		PERMISSÕES DE BANCO		
sco		arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)		

Tabela sco_funcionarioareaabrangencia				
Descrição: Tabela que relaciona os dados das tabelas sco_areaabrangencia e sco_funcionario				
Campos sco_funcionarioareaabrangencia				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
faa_id	SERIAL	Número seqüencial para identificar a relação funcionário e área de abrangência	x	
fun_id	INTEGER	Código do funcionário		x (com a coluna fun_id em sco_funcionario)
aab_id	INTEGER	Código da área de abrangência		x (com a coluna aab_id em sco_areaabrangencia)
Permissões sco_funcionarioareaabrangencia				
USUÁRIO		PERMISSÕES DE BANCO		
sco		arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)		

Tabela sco_contato				
Descrição: Tabela contendo dados referentes ao contato.				
Campos sco_contato				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
con_id	SERIAL	Número seqüencial para identificar o contato	x	
sol_id	INTEGER	Código do solicitante		x (com a coluna sol_id em sco_solicitante)
con_horariocontato	Varchar(255)	Horário para contato do contato		
Permissões sco_contato				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_grauurgencia				
Descrição: Tabela contendo dados referentes ao grau de urgência da ocorrência.				
Campos sco_grauurgencia				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
gur_id	SERIAL	Número seqüencial para identificar o grau de urgência	x	
gur_nome	Varchar(255)	Nome do grau de urgência		
gur_descricao	Varchar(255)	Descrição do grau de urgência		
Permissões sco_grauurgencia				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_formaatendimento				
Descrição: Tabela contendo dados referentes à forma de atendimento.				
Campos sco_formaatendimento				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
fat_id	SERIAL	Número seqüencial para identificar a forma de atendimento	x	
fat_nome	Varchar(255)	Nome da forma de atendimento		
fat_descricao	Varchar(255)	Descrição da forma de atendimento		
Permissões sco_formaatendimento				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_pesquisaqualidade				
Descrição: Tabela contendo dados referentes a pesquisa de qualidade.				
Campos sco_pesquisaqualidade				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
pqu_id	SERIAL	Número seqüencial para identificar a pesquisa de qualidade	x	
pqu_nome	Varchar(255)	Nome da pesquisa de qualidade		
pqu_descricao	Varchar(255)	Descrição da pesquisa de qualidade		
toc_id	INTEGER	Código do tipo de ocorrência		x (com a coluna toc_id em sco_tipoocorrencia)
Permissões sco_pesquisaqualidade				
USUÁRIO	PERMISSÕES DE BANCO			
sco	arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)			

Tabela sco_pergunta				
Descrição: Tabela contendo dados referentes às perguntas das pesquisas de qualidade.				
Campos sco_pergunta				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
pgu_id	SERIAL	Número seqüencial para identificar a pergunta	x	
pgu_pergunta	Varchar(255)	Descrição da pergunta		
pgu_numero	INTEGER	Número da pergunta		
pqu_id	INTEGER	Código da pesquisa de qualidade		x (com a coluna pqu_id em sco_pesquisaqualidade)
Permissões sco_pergunta				
USUÁRIO		PERMISSÕES DE BANCO		
sco		arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)		

Tabela sco_resposta				
Descrição: Tabela contendo dados referentes às respostas das perguntas das pesquisas de qualidade.				
Campos sco_resposta				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
res_id	SERIAL	Número seqüencial para identificar a resposta	x	
res_resposta	Varchar(255)	Descrição da resposta		
res_numero	INTEGER	Número da resposta		
pgu_id	INTEGER	Código da pergunta		x (com a coluna pgu_id em sco_pergunta)
Permissões sco_resposta				
USUÁRIO		PERMISSÕES DE BANCO		
sco		arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)		

Tabela sco_perfil				
Descrição: Tabela contendo dados referentes ao perfil dos usuários				
Campos sco_perfil				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
per_id	SERIAL	Número seqüencial para identificar o perfil	x	
per_nome	Varchar(255)	Nome do perfil		
per_descricao	Varchar(255)	Descrição do perfil		
Permissões sco_perfil				
USUÁRIO		PERMISSÕES DE BANCO		
sco		arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)		

Tabela sco_fluxoocorrencia				
Descrição: Tabela contendo dados referentes ao fluxo das ocorrências				
Campos sco_fluxoocorrencia				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
flu_id	SERIAL	Número seqüencial para identificar o fluxo da ocorrência	x	
oco_id	INTEGER	Código da ocorrência		x (com a coluna oco_id em sco_ocorrencias)
wrk_id	INTEGER	Código do workflow		x (com a coluna wrk_id em sco_workflow)
tas_id	INTEGER	Código da tarefa corrente		x (com a coluna tas_id em sco_tarefa)
flu_checkedout	Boolean	Se a tarefa atual está em check out		
flu_tarefa correntefinalizada	Boolean	Se a tarefa atual está finalizada		
flu_finalizado	Boolean	Se o fluxo foi finalizado		
Permissões sco_fluxoocorrencia				
USUÁRIO		PERMISSÕES DE BANCO		
sco		arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)		

Tabela sco_solucaoocorrencia				
Descrição: Tabela contendo dados referentes a solução das ocorrências				
Campos sco_solucaoocorrencia				
NOME DO CAMPO	TIPO DE DADO	DESCRIÇÃO	PK	FK
slu_id	SERIAL	Número seqüencial para identificar a solução da ocorrência	x	
oco_id	INTEGER	Código da ocorrência		x (com a coluna oco_id em sco_ocorrencias)
fun_id	INTEGER	Código do funcionário responsável pela solução		x (com a coluna fun_id em sco_funcionario)
slu_solucao	Varchar(255)	Solução da ocorrência		
slu_datahora	Date	Data e hora da ocorrência		
Permissões sco_solucaoocorrencia				
USUÁRIO		PERMISSÕES DE BANCO		
sco		arwd (ATUALIZAÇÃO, LEITURA, ESCRITA, e EXCLUSÃO)		

APÊNDICE IV – CÓDIGO-FONTE DA APLICAÇÃO

O código-fonte da aplicação foi disponibilizado em CD anexo a este projeto.